HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

qutebrowser made extensible

Term Project (Studienarbeit)

Autumn Term 2018/2019

Department of Computer Science
University of Applied Sciences Rapperswil (HSR)
`www.hsr.ch`

Author: Florian Bruhin

Advisor: Prof. Stefan Keller, HSR

Date: December 21, 2018

Abstract

The qutebrowser project is a web browser, comparable to Google Chrome or Mozilla Firefox, which is focused on keyboard usage and having a minimal user interface. It is aimed at power-users who value customizability and efficiency, but are willing to accept a rather steep learning curve compared to “traditional” web browsers.

In contrast to Chrome and Firefox, qutebrowser did not support extending its functionality via extensions. Over its lifetime, various features have been added to its core by its maintainer and contributors. However, this caused the core to grow substantially, becoming increasingly complex over time.

Many users of qutebrowser have specific feature requests and workflows. It should be possible for them to extend qutebrowser in an easy way in order to keep the core small and simple. The goal of this project was to make qutebrowser extensible by introducing a clearly defined API which can be used to develop extensions.

Before attempting to expose such an API from qutebrowser, various problematic areas in its codebase had to be cleaned up due to "technical debt" accumulating in the past. Subsequently, functionality suitable for moving out of the core into extensions was identified. Based on the selected areas of code, a concept for an extension API was developed. After implementing said API, large parts of qutebrowser’s functionality could be moved into extensions. The resulting changes increased code maintainability and simplicity.

More information about qutebrowser can be found on its project website:
`https://www.qutebrowser.org/`.

Management Summary

Introduction

The qutebrowser project is a web browser, comparable to Google Chrome or Mozilla Firefox, which is focused on keyboard usage and having a minimal user interface. It is aimed at power-users who value customizability and efficiency, but are willing to accept a rather steep learning curve compared to “traditional” web browsers.

Since qutebrowser uses the Python programming language in conjunction with the Qt library for graphical user interfaces, it is standing on the shoulders of giants: It does not implement complex tasks such as downloading and executing HTML/CSS/JavaScript code itself. Instead, it relies on the QtWebEngine project to do so, which is largely based on the same code as Google Chrome.

Work on qutebrowser started in December 2013. Since then, it has gained a big community of thousands of users and dozens of contributors.

Figure 1.: Technologies used

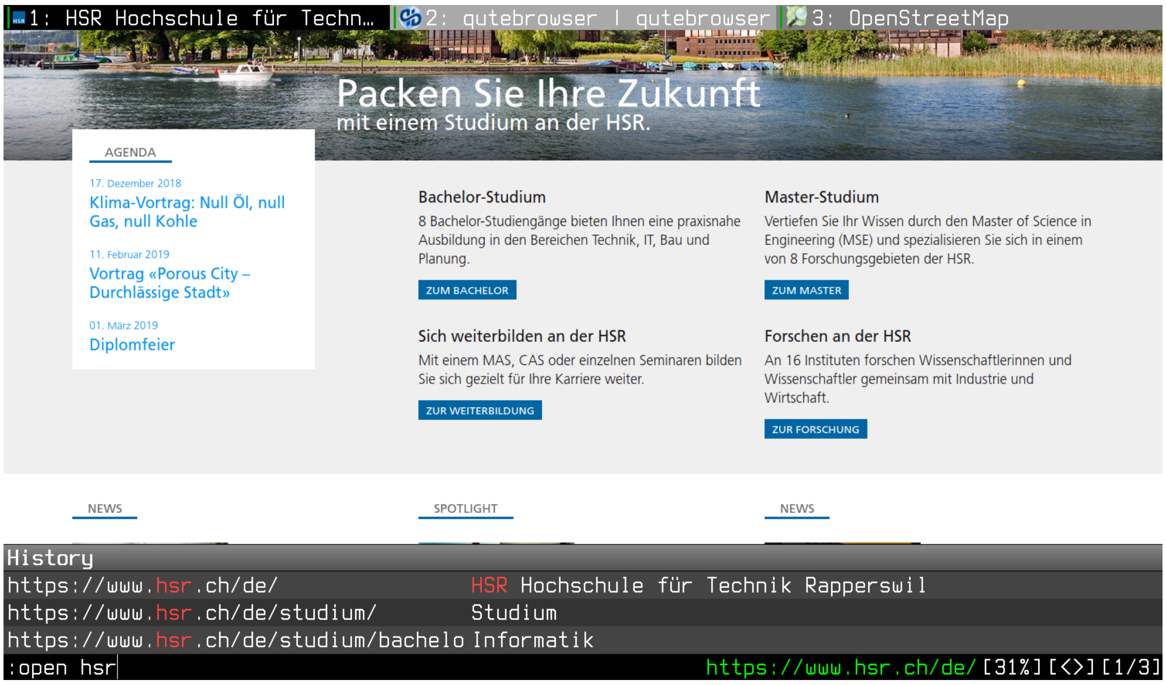
Objective

In contrast to Chrome and Firefox, qutebrowser did not support extending its functionality via extensions. Over its lifetime, various features have been added to its core by its maintainer (who is the author of this report) and contributors. However, this caused the core to grow substantially, becoming more and more complex over time.

Many users of qutebrowser are power-users and thus have specific (and sometimes unique) feature requests and workflows. It should be possible for those users to extend qutebrowser with custom extensions in an easy way in order to keep the core small and simple.

The goal of this project was to make qutebrowser extensible by introducing a clearly defined application programming interface (API) which can be used to develop extensions.

Since qutebrowser already has a thriving community, this change also intends to decentralize future development efforts, as it enables users and developers to maintain their extensions independently from qutebrowser's core.



17. Dezember 2018
Klima-Vortrag: Null Öl, null Gas, null Kohle

11. Februar 2019
Vortrag «Porous City – Durchlässige Stadt»

01. März 2019
Diplomfeier

Bachelor-Studium
8 Bachelor-Studiengänge bieten Ihnen eine praxisnahe Ausbildung in den Bereichen Technik, IT, Bau und Planung.

Master-Studium
Vertiefen Sie Ihr Wissen durch den Master of Science in Engineering (MSE) und spezialisieren Sie sich in einem von 8 Forschungsgebieten der HSR.

Sich weiterbilden an der HSR
Mit einem MAS, CAS oder einzelnen Seminaren bilden Sie sich gezielt für Ihre Karriere weiter.

Forschen an der HSR
An 16 Instituten forschen Wissenschaftlerinnen und Wissenschaftler gemeinsam mit Industrie und Wirtschaft.

```
History
https://www.hsr.ch/de/                HSR Hochschule für Technik Rapperswil
https://www.hsr.ch/de/studium/        Studium
https://www.hsr.ch/de/studium/bachelo Informatik
:open hsr                              https://www.hsr.ch/de/ [31%] [<>] [1/3]
```

Figure 2.: qutebrowser displaying its history completion

Procedure / Results

In order to follow best practices in the software development world, tools for checking data types were evaluated. The *mypy* tool is now run regularly over the code which resulted in various lingering defects being found in qutebrowser itself and in related projects. Important parts of qutebrowser were annotated with type information in order to prevent such issues in the future, also serving as additional developer documentation.

Before attempting to expose an interface for extensions from qutebrowser, various problematic areas in qutebrowser's codebase had to be cleaned up due to "technical debt" accumulating in the past. Subsequently, functionality suitable for moving out of the core into extensions was identified. Based on the selected areas of code, a concept for an extension API was developed. After implementing said API, large parts of qutebrowser's functionality (such as the adblocker, blocking advertisements on websites) could be moved into extensions. The resulting changes increased code maintainability and simplicity.

Future Work

This research project was focused on increasing qutebrowser's software quality and moving parts of its core into extensions. As a next step, additional functionality should be exposed to extensions, allowing more code to be moved out of the core.

When the exposed interfaces are expected to be reasonably stable, they should then be gradually opened to interested external developers so that third-party extensions can be written.

After third-party extensions gain enough traction, an "extension manager" should be developed so that users can easily install and update available extensions via a graphical user interface.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

1.1. Background

The qutebrowser project is a web browser, comparable to Google Chrome or Mozilla Firefox, which is focused on being keyboard-driven and having a minimal user interface. It is aimed at power users who value customizability and efficiency, but are willing to accept a rather steep learning curve compared to "traditional" web browsers.

Its philosophy and keybindings are inspired by the Vim editor[1] which is available on almost any UNIX system (such as macOS or Linux). Similar projects exist as addons for Firefox (such as Tridactyl[2]) or Chrome (Vimium[3]). However, due to being constrained by the extension API exposed by Firefox and Chrome, those projects lack various features which qutebrowser offers.

Since qutebrowser uses the Python programming language in conjunction with the Qt library for graphical user interfaces, it is standing on the shoulders of giants: It does not implement complex tasks such as downloading and executing HTML/CSS/JavaScript code itself. Instead, it relies on the QtWebEngine project to do so, which is largely based on the same code as Google Chrome.

An extension API for users to write their own extensions to qutebrowser is a long-standing feature request[4], which has often been requested by its users.

The original inspiration for qutebrowser was the *dwb* project[5] which is a web browser with a very similar philosophy. The author of qutebrowser used dwb for a long time, but it became apparent that dwb's author lacked the time and interest to continue working on it, thus many dwb users were looking for a viable alternative. Since no suitable alternative existed, work on qutebrowser started in December 2013, after which it quickly gained traction.

It is difficult to estimate qutebrowser's user count, but it is most likely used by a couple thousand users, so an extension API is also vital in order to be able to move less popular features out of the core part of qutebrowser.

There are two backends (rendering engines) which can be used with qutebrowser:

- *QtWebKit* which is based on the WebKit[6] project.
- *QtWebEngine* which is based on the Chromium[7] project and used by default. Chromium is the open source project behind Google Chrome.

[1] `https://www.vim.org/`
[2] `https://github.com/tridactyl/tridactyl`
[3] `https://vimium.github.io/`
[4] `https://github.com/qutebrowser/qutebrowser/issues/30`
[5] `https://portix.bitbucket.io/dwb/`
[6] `https://www.webkit.org/`
[7] `https://www.chromium.org/`

In order to allow using either backend, qutebrowser provides an abstraction layer over the two libraries, implementing the adapter pattern (Johnson et al. 2005, 139ff). This abstraction layer is referred to as "tab API", and documented in section 4.3.

1.2. Vision

Many qutebrowser users are power-users and, as such, have very specific (and sometimes unique) feature requests and workflows. It should be possible for those users to extend qutebrowser with custom extensions in an easy way, in order to keep qutebrowser's core small.

Since qutebrowser already has a thriving community, this change also intends to decentralize development efforts, as it enables users and developers to maintain their extensions independently from the core development.

1.3. Objectives

The official task description ("Aufgabenstellung") in German can be found in appendix B.

1.3.1. Merging contributions

When this project started, various third-party contributions were pending a code review, due to being ignored during the exam session preceding this semester.

Since those contributions would conflict with the refactorings needed before work on the extension API can start, those should be reviewed and merged (or rejected) at the beginning of this project.

1.3.2. Refactorings

Initially, qutebrowser was developed without knowledge of proper software engineering practices, which resulted in some maintainability issues. While many of those issues have since been resolved, some still remain. The refactorings needed to fix those issues will affect the API exposed to extensions, and therefore should be taken care of before attempting to design an extension API.

The following changes are planned:

- `https://github.com/qutebrowser/qutebrowser/issues/1456`:
  Parts of qutebrowser already use Python type annotations[8], but only if contributors decide to use them. In addition to that, no type checker such as mypy[9] is currently run as part of qutebrowser's continuous integration (CI) pipeline, thus allowing regressions to occur. As part of this project, a type checker should be introduced into the CI infrastructure, and any code exposed via the extension API should be annotated with proper type annotations.

[8] `https://www.python.org/dev/peps/pep-0484/`
[9] `http://mypy-lang.org/`

- `https://github.com/qutebrowser/qutebrowser/issues/345`:
  To generate HTML documentation, qutebrowser currently uses asciidoc[10] which is unsuitable for API documentation and ceased maintenance. The Sphinx[11] tool should be introduced instead, which is tailored to documenting APIs.

- `https://github.com/qutebrowser/qutebrowser/issues/640`:
  Global objects are registered in a object registry based on a name as string ("stringly-typed"[12]). This historically caused various object-lifetime related issues, and also breaks tooling such as the mypy type checker. All code using the object registry should be refactored to use better alternatives such as constructor arguments (dependency injection).

1.3.3. Extension API

After the needed refactorings are finished, an extension API for qutebrowser should be designed and implemented. In order to design the API with real-world usages in mind, the following steps should be taken:

- Identify what functionality to expose over the extension API. This can be done by analyzing code in qutebrowser's core which is suitable for moving into extensions. Use cases for future third-party extensions should be considered as well.
- Research existing APIs for browser extensions and analyze prior solutions.
- Design and implement a suitable API based on the knowledge gained.
- Use the newly added API by moving code out of qutebrowser's core into extensions (which are shipped with qutebrowser).
- Review the finished API to make sure it is consistent and intuitive.

1.3.4. Documentation

As shown in GitHub's "Open Source Survey" (GitHub, Inc. 2017), good documentation is highly valued in open source projects, but also is frequently overlooked by project maintainers.

Therefore, good documentation is an explicit goal of this project: The resulting extension API should be documented well by providing API documentation, usage examples, and an example extension.

Additionally, this report is written in English, allowing a bigger audience to read it. Thus, future extension authors or qutebrowser contributors can read this documentation to get further background information about its extension API.

[10] `http://asciidoc.org/`
[11] `http://www.sphinx-doc.org/`
[12] `http://wiki.c2.com/?StringlyTyped`

1.4. Context

The software and version constraints are mostly given by the existing project:

- Python[13] 3 (3.5 or newer)
- Qt[14] 5 (5.7 or newer), used via PyQt5[15]
- pytest[16] as test framework
- Various code quality tools: pylint[17], flake8[18] and others

As qutebrowser is a pre-existing project with a vibrant community, external contributions are expected to continue (despite an announcement[19] asking people to refrain from making larger contributions). This can be challenging, as it results in refactorings being carried out against a moving target. Because of the nature of open-source contributions, it is hard to foresee or control which areas external contributors are changing. At the beginning of the SA, some time was allocated for merging external contributions (pull requests) which were already open. For the rest of the SA, such contributions will be dealt with on a best effort basis, with the main focus being this documentation and the work required for the extension API.

1.4.1. Notable language features

This document assumes that the reader is familiar with a programming language following the object oriented paradigm and related concepts such as classes.

Some sections refer to concepts which are specific to Python or Qt and are thus explained in more detail below.

Python decorators

Python allows to annotate a function or method with a "decorator". As an example, this can be used to define properties:

```python
class Example:

    @property
    def prop(self):
        return 2 + 2

ex = Example()
print(ex.prop)  # 4
```

[13] `https://www.python.org/`
[14] `https://www.qt.io/`
[15] `https://www.riverbankcomputing.com/software/pyqt/intro`
[16] `https://pytest.org/`
[17] `https://pylint.org/`
[18] `http://flake8.pycqa.org/`
[19] `https://lists.schokokeks.org/pipermail/qutebrowser-announce/2018-October/000053.html`, accessed 2018-11-12

The same feature can also be used to implement custom decorators. Functions are first-level objects in Python, thus a decorator receives the function it decorates as an argument and returns a function (which may be modified).

A more thorough explanation of decorators can be found here:
`https://realpython.com/primer-on-python-decorators/`

Python modules and packages

A Python module simply is a file with a `.py` extension containing Python code. A package is a folder containing modules and a special `__init__.py` file (often empty) as a package marker.

Qt signals and slots

Signals and slots are an important feature of the Qt library to allow independent objects to communicate with each other. They can be considered an abstraction of the observer pattern (Johnson et al. 2005, 293ff), similar to delegates and events in C#.

A class can define a signal which it uses to notify other objects (e.g. about changed state):

```python
class Example(QObject):

    text_changed = pyqtSignal(str)
```

A slot is a function or method which can be connected to a signal:

```python
@pyqtSlot(str)
def on_text_changed(text: str) -> None:
    print(text)
```

Signals provide a `connect()` method to connect them to a slot:

```python
ex = Example()
ex.text_changed.connect(on_text_changed)
```

as well as an `emit()` method used to call all slots connected to it – the above `Example` class could have a `set_text` method like this:

```python
def set_text(self, text: str) -> None:
    # ...
    self.text_changed.emit(text)
```

More information about Qt signals and slots can be found in its (C++) documentation:
`https://doc.qt.io/qt-5/signalsandslots.html`

1.5. Methods and Structure

1.5.1. Structure of this document

This document is organized in roughly chronological order.

In this chapter, the background necessary to understand the project and its goals is explained. Chapter 2 documents the project methodology used and allocates the time available. It also analyzes the risks involved. Chapter 3 specifies the project requirements in more detail. In chapter 4, existing similar work is examined. Chapter 5 notes the criteria being considered for the project. Based on these criteria, a concept was designed in chapter 6. Finally, chapter 7 explains the actual implementation, while chapter 8 analyzes the results.

In appendix A, a glossary of the terms and abbreviations used can be found.

1.5.2. Design decisions

In order to take informed and sustainable design decisions, the *Y-Approach* (Zimmermann 2012) was evaluated. It proposes the following template for design decisions:

- In the context of *use case uc and/or component co*,
- ... facing *non-functional concern c*,
- ... we decided for *option o_1*
- and neglected *options o_2 to o_n*,
- ... to achieve *quality q*,
- ... accepting downside *consequence c*.

The template serves as a good reminder of the aspects to keep in mind while taking a decision. However, it was found to not be very helpful for documentation, as a different structure (such as a list of considerations, or prose text) is often more readable. Thus, it was used as an inspiration only, but the explanations in this document do not strictly follow its structure.

Finally, note that the Y-Approach is intended for bigger architectural decisions. However, since the high-level architecture of qutebrowser and its extension API is already predetermined, only design decisions related to the API design and its implementation remain.

1.5.3. Tools used

Since this project was realized by a single author (rather than in a group), project management tooling was kept to a minimum. GitHub issues and its project board features were used for project management, along with Clockify[20] for time tracking.

This document was typeset in LaTeX using the *Computer Modern Bright* font and various additional packages. GNU Emacs (with vim keybindings) was used as text editor.

[20]`https://www.clockify.me/`

2. Project Management

2.1. Releases and Milestones

Since this project is done without any external industry partner, there is no immediate need for fixed releases or prototypes during the SA. However, the following releases are planned:

- A v1.5.0 feature release as soon as PyQt 5.11.3 is released upstream, with v1.5.x patch releases as needed in case of regressions or serious enough bugs.
- v1.6.0 after the work on refactoring qutebrowser's core into extensions is completed, in order to get the changes out to users as soon as possible. Afterwards, v1.6.x patch releases as needed (likely after the SA is finished).
- In the future, after the extension API is open for third-party extensions, a v2.0.0 release with some other big changes (like dropping support for the older QtWebKit backend).

At the end of week 11, an API review milestone is planned, where the extension API design gets reviewed by employees of the IFS (Institute for Software) at the HSR. At the end of week 12, feature freeze comes into effect and the focus is shifted to bugfixes and documentation.

2.2. Team and Roles

Florian Bruhin is both the primary maintainer of qutebrowser and the sole author of this student research project. He has been working on qutebrowser since December 2013 and started studying Computer Science at HSR in 2016.

Joël Schwab intended to co-author this research project, but unfortunately did not pass some exams, which were a precondition to be allowed to do the SA project this semester.

Professor *Stefan Keller*, institute partner at the Institute for Software (IFS) at HSR is the advisor for this project. *Raphael Das Gupta*, research assistant at the IFS, helped out with code reviews and architectural decisions.

The qutebrowser community is not directly involved in this research project, but is the primary audience of the resulting work. It has also contributed many ideas and use cases for future extensions[1]. There is no external industry partner.

Fritz Reichwald (fiete201[2]) is a long-time qutebrowser user who started working on migrating qutebrowser's documentation system from asciidoc to Sphinx (see section 1.3.2). However, he was unable to finish his work in time, so all API documentation work mentioned in this documentation is the author's own.

[1]`https://github.com/qutebrowser/qutebrowser/issues/30`
[2]`https://github.com/fiete201`

2.3. Process Model

Because of previous experience as part of HSR's "Engineering Project", the "Scrum+" process model will be used, i.e., an agile Scrum process with an additional "End of Elaboration" milestone. At the beginning of the SA, there was a phase with the goal of merging existing third-party contributions (pull requests) into qutebrowser, so that the changes required for extensions do not conflict with existing work. Thus, an additional *preparation* phase has been introduced, resulting in the following phases: Inception, Preparation, Elaboration, Construction, Transition.

2.4. Project Schedule

8 ECTS credits are awarded for the SA course. Since one ECTS point is equivalent to a workload of 25–30 hours (Hochschulrat 2015) and this SA is written by a single author, this amounts to 200–240 hours of work. The semester consists of 14 weeks, thus, an average workload of 14–17 hours per week is expected. This time is allotted as shown in figure 3.

Due to the relatively long *preparation* phase at the beginning of the project, the *construction* phase gets unusually short. This is unfortunate but unavoidable – it is important to integrate existing contributions before starting to refactor the existing code, at least for third-party changes which would lead to conflicts with refactoring changes.

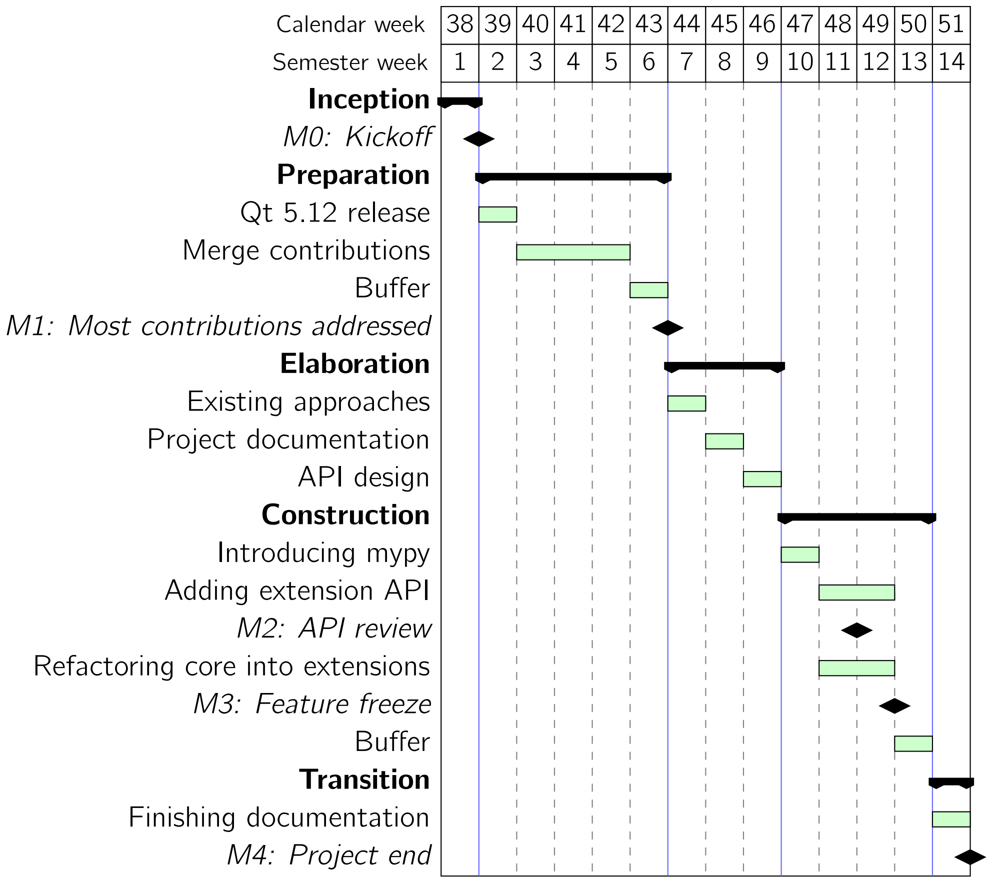


Figure 3.: Project schedule

2.5. Risks

The following risks have been identified at the beginning of this project:

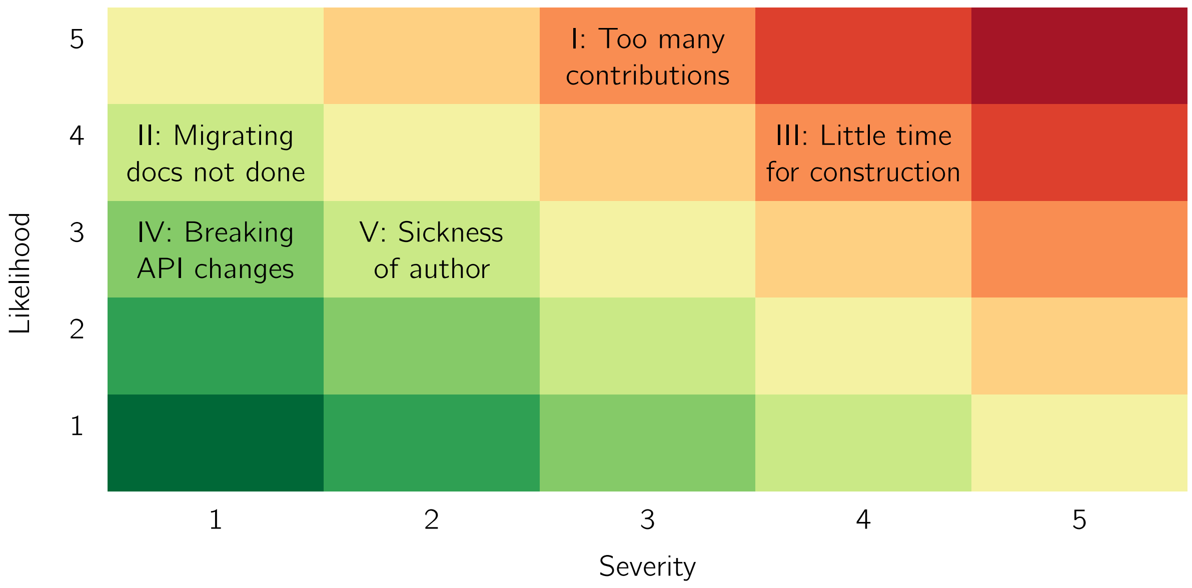


Figure 4.: Risk matrix

The following measures have been taken to mitigate those risks:

Risk	Description	Mitigation
I	Too many external contributions	Asking contributors to hold back further contributions; ignoring existing contributions
II	Migrating documentation toolchain to Sphinx (done by external contributor) is not done in time	Initial extension API documentation can be separate from qutebrowser documentation
III	Little time in construction phase	Work on extension API is very scalable, buffers in schedule
IV	Future breaking API changes needed	API is not exposed for third-party contributions yet
V	Sickness of author	Work on extension API is very scalable

Table 2.1.: Risk mitigations

Since some of these risks are hard to quantify in hours lost (such as "too many external contributions"), a probability between 1 and 5 has been assigned instead.

As explained in section 1.4, it is expected that third-party contributors continue to submit changes (in the form of pull requests) while this SA is ongoing. An announcement[3] was sent out, asking contributors to cut back external contributions. However, a similar announcement was sent out during past exam seasons, with mixed results – many contributors continue to submit changes regardless. Initially, there is a phase focused on merging existing contributions (see the project schedule in figure 3). Afterwards, contributions are dealt with on a best effort basis, being ignored if they are non-trivial.

There is little time in the construction phase compared to other projects. Additionally, this SA is written by a single author. Thus, sickness or other obstructions have a bigger than usual impact. This is counteracted by trying to keep the scope of the SA clear, but flexible. Additionally, buffer time is added after the preparation and construction phases in the project schedule (see figure 3).

If some preparatory work is not fully finished, it is possible that upcoming changes cause incompatible changes in the extension API. However, this SA is focused on moving core code into extensions, rather than providing an extension API for third-party extensions. Therefore, it is still possible to make those changes after the SA is finished, without breaking third-party code.

[3] `https://lists.schokokeks.org/pipermail/qutebrowser-announce/2018-October/000053.html`, accessed 2018-11-12

3. Requirements Specification

3.1. Use Cases

This project extends an existing codebase with an extension API rather than starting a new project from scratch. By its nature, it is difficult to predict how an extension API will be used in the future. Because of that, an use-case diagram would not adequately describe the motivation for the extension API.

Various ideas for future third-party extensions have been voiced by the qutebrowser community; they are collected in a GitHub issue[1]. However, the main aim of this project (and thus the main focus for the extension API) is reducing the complexity of qutebrowser's core.

3.2. Non-functional Requirements

The following non-functional requirements are relevant for qutebrowser's extension API:

Security The security model used for qutebrowser extensions assumes that extensions are trusted, i.e., may run arbitrary code. See section 6.3 for a more detailed explanation.

Simplicity It should be trivial for a user to extend her qutebrowser setup with a custom extension. Thus, getting started with writing a third-party extension should be as straightforward as possible, without requiring packaging multiple files into a custom format. Also see section 4.2.1 for an explanation on how this topic is handled for WebExtensions, and how qutebrowser's API differs from that.

Learnability When the extension API is opened for third-party contributions, it should be easy to get accustomed to it. Therefore, the API should be intuitive for Python programmers, and well documented.

Compability The qutebrowser project runs on a variety of different software versions – various operating systems are supported (Linux, macOS, Windows, and more), including different Qt and Python versions. The extension API should abstract over these differences, so an extension written for qutebrowser (or core code moved into such an extension) runs in all situations qutebrowser itself can run.

Performance Having code in extensions (rather than in the core part of a project) can result in performance degradations due to more moving parts and more layers being involved. Since those impacts are usually small individually (but might be big enough to be relevant collectively), no relevant impact is expected in the scope of this SA. However, performance considerations might be a good reason to keep some code in the core rather than moving it to extensions in the future.

[1] `https://github.com/qutebrowser/qutebrowser/issues/30`, accessed 2018-11-08

4. Existing APIs

Other browsers (such as Firefox or Chrome) have been supporting extensions for a long time. When designing an extension API, it is useful to understand this previous work. It can serve as an inspiration, in order to follow common standards and learn from mistakes made in the past. Therefore, various existing extension APIs have been analyzed.

4.1. Firefox XUL extension API

Older versions of the Firefox web browser used to have a very powerful extension API, based on its XUL (XML User Interface Language) technology. However, this approach presented various challenges and was thus recently abandoned, while adopting the WebExtensions standard.

The apparent philosophy behind “legacy” Firefox addons was to allow maximum customizability from extensions – however, this came with various drawbacks which ultimately led to Mozilla abandoning that approach.

The motivations to deprecate and subsequently remove the legacy addon API listed in Mozilla’s blog post (Needham 2015) were as follows:

- Chrome and Opera (and nowadays also Microsoft Edge) already supported the WebExtension API, so a switch to the WebExtension API would drastically reduce the effort required for developers when implementing extensions with cross-browser compatibility: *“We would like add-on development to be more like Web development: the same code should run in multiple browsers according to behavior set by standards, with comprehensive documentation available from multiple vendors.”*

- Firefox’ Electrolysis (e10s) project[1] was a big change in its codebase, with the goal of separating tabs into separate processes, for security and performance reasons. Many legacy addons were not compatible with the changes necessary for Electrolysis. This forced either the add-on developer to make (sometimes intricate) changes to their code; or the user’s Firefox instance to run in a special fallback mode: *“Add-ons that haven’t been upgraded to work with Electrolysis will run in a special compatibility environment that resembles single-process Firefox as much as possible. [...] However, [the fallback is] much slower than the equivalent DOM operations in single-process Firefox, and can affect the user experience negatively. Also, some accesses aren’t supported by the compatibility layer and will throw exceptions.”*

- Since Firefox is a quite popular product, malicious Firefox addons started to become an attractive attack vector for bad actors. With legacy addons, addon code is able to run arbitrary code and freely modify Firefox internals on the user’s machine, which turns untrusted addons into a security liability (Veditz 2015).

[1] `https://wiki.mozilla.org/Electrolysis`

- Legacy addons hindered Firefox development in general, since its powerful addon API introduced a tight coupling between Firefox' internal code and the code in third-party addons: *"A permissive add-on model means that we have limited flexibility in changing the foundations of Firefox. [...] Without a fundamental shift to the way Firefox add-ons work, we will be unable to use new technologies like Electrolysis, Servo*[2] *or browser.html*[3] *as part of Firefox. The tight coupling between the browser and its add-ons also creates shorter-term problems for Firefox development. It's not uncommon for Firefox development to be delayed because of broken add-ons."*

While qutebrowser should learn from the mistakes made in Firefox' legacy API, a more thorough analysis of the XUL API design proved to be difficult. Archived API documentation is still available[4], but bad documentation was one of the criticisms of XUL addons (Needham 2015). Since the API is not in active use anymore, and ties into Firefox' core code deeply, no further analysis was performed.

The key takeaway for qutebrowser is that it should have a minimal and clearly outlined extension API, rather than naively exposing its internal Python code to extensions.

4.2. WebExtensions API

Currently, there are ongoing efforts towards a *WebExtensions* API for browser extensions, which is shared between different browsers. WebExtensions are supported by Chrome, Opera, Firefox and Edge. Efforts are currently underway to standardize the API as a W3C specification (Pietraszak 2017). At the moment, each browser has a slightly different set of supported APIs, with some divergence in naming. As an example, the `chrome.` module is used in Chromium for browser-specific features, while Firefox uses `browser.` instead.

If qutebrowser supported the WebExtension API, it would follow a common standard and enable running thousands of existing Chromium extensions with little to no adjustment to their code.

Unfortunately, WebExtensions are unsuitable for qutebrowser, for various reasons:

- Full WebExtension support would likely require some degree of support by the underlying rendering engine. While there is an open suggestion[5] in Qt's bug tracker, no work on that has been started so far.

- Implementing WebExtension support in Python only (without any support from the underlying backend) is hard, if possible at all. Neither backend allows deep access to its JavaScript engine (V8[6] for QtWebEngine, JavaScriptCore for QtWebKit), which means extension code would have to run in a separate JavaScript interpreter (such as Qt's `QJSEngine`[7] class). However, extensions should have access to web contents. Thus, a secure communication channel is needed between the backend's JavaScript interpreter (used for the page), and the separate interpreter used for extensions. In addition, it should only be possible for extensions to access page content, not the other way around, as that

[2]An experimental new rendering engine by Mozilla, implemented in the Rust programming language. Parts of Servo (such as its CSS renderer) have since been merged into Firefox.
[3]A Mozilla research project which implements a browser completely in HTML, now retired.
[4]`https://developer.mozilla.org/en-US/docs/Archive/Add-ons`
[5]`https://bugreports.qt.io/browse/QTBUG-61676`
[6]`https://v8.dev/`
[7]`http://doc.qt.io/qt-5/qjsengine.html`

would be a security issue. All in all, this approach is prohibitively complex, and thus out of scope for this SA.

- Most qutebrowser users and contributors are accustomed to writing Python code (since qutebrowser itself is written in Python). While a WebExtension API (in JavaScript) would allow using thousands of existing Chrome extensions, a Python API will make it easier for users to write custom additions to qutebrowser to suit their needs.
- The security model of WebExtensions is fundamentally different from what is intended for qutebrowser extensions – WebExtension code is sandboxed, and unable to run arbitrary code on the user's machine. This can be worked around by using WebExtension's "native messaging"[8] capability with a separate application running in the background, but doing so is cumbersome. See 6.3 for an explanation on the security model of qutebrowser extensions.
- The WebExtension API was not designed with the Qt APIs in mind, yet qutebrowser is bound to using those. In some cases it might be possible to write an adapter to bring the two approaches together; but if they are fundamentally different, doing so might prove difficult. With a custom API, there is more control over the tradeoff between an API which closely follows Qt's APIs (and thus reduces friction and complexity), or an API which prioritizes other considerations (see section 5.1).

While it is not possible for qutebrowser to implement support for WebExtensions, they are still useful as a source for API design inspiration. However, it should be noted that the WebExtension API is closely tied to the JavaScript language, so architecture decisions taken there will not necessarily be applicable to qutebrowser's Python extension API.

4.2.1. Anatomy of a WebExtension

As explained in Mozilla's documentation[9], a WebExtension consists of various files:

- A `manifest.json` manifest file, which lists meta information about an extension, such as its name, its author, or the permissions it requests.
- Background scripts, which are used to implement long-running operations. An extension's background scripts are loaded for the entire lifetime of an extension, until it is disabled or uninstalled.
- Content scripts, which are injected into loaded web pages, and can interact with their content (access and manipulate the DOM). They also have some additional permissions, which scripts supplied by the page do not have, such as messaging with an extension's background scripts.
- HTML pages for user interfaces such as an extension's option page or sidebar.
- Other resources such as icons.

[8]`https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging`
[9]`https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension`, accessed 2018-11-08

These files are then packaged into a specially named ZIP file (XPI in case of Firefox; CRX for Chromium) and distributed via an extension store (a special website run by Mozilla/Google).

For qutebrowser extensions, a simpler approach will be taken: Extensions can consist of a single standalone `.py` file which implements the necessary functionality. This mechanism is inspired by the plugin handling of the pytest project[10]. In the future, this makes it trivial for users to add their own extensions to qutebrowser. That solution is also very straightforward and appropriate for the initial goal of refactoring qutebrowser's core code into extensions.

Due to extensions being written in Python, the above separation between content and background scripts also does not fully apply to qutebrowser's API. The Python extensions can be seen as a background script.

4.2.2. Analysis of the WebExtension API

The WebExtensions API is separated into various modules, each of which is briefly analyzed in the context of qutebrowser in this section, sorted alphabetically. The indented descriptions are copied verbatim from Mozilla's API documentation[11].

alarms

> Schedule code to run at a specific time in the future.

Since qutebrowser extensions will have access to the Qt GUI library, no equivalent to this module is needed. Qt provides the `QTimer`[12] class, which can be used for equivalent functionality.

bookmarks

> The WebExtensions bookmarks API lets an extension interact with and manipulate the browser's bookmarking system. You can use it to bookmark pages, retrieve existing bookmarks, and edit, remove, and organize bookmarks.

Bookmarks in qutebrowser are currently much simpler than they are in Firefox or Chrome: They are not organized in a tree-like structure, and features like tags are missing. Therefore, the bookmarks WebExtension API does not fit well with qutebrowser bookmarks.

There is an ongoing contribution[13] to add such features, and it would make little sense to add an extension API before that contribution is merged. Since access to bookmarks is not vital for an extension API, it is currently out of scope.

[10] `https://docs.pytest.org/en/latest/writing_plugins.html`, accessed 2018-11-08
[11] `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API`, accessed 2018-11-02
[12] `http://doc.qt.io/qt-5/qtimer.html`
[13] `https://github.com/qutebrowser/qutebrowser/pull/3855`

browserAction, menus, pageAction, sidebarAction

browserAction: Adds a button to the browser's toolbar.

menus: Add items to the browser's menu system.

pageAction: A page action is a clickable icon inside the browser's address bar.

sidebarAction: Gets and sets properties of an extension's sidebar.

These modules allow extensions to add their own user interface elements to the browser, as shown in figure 5.

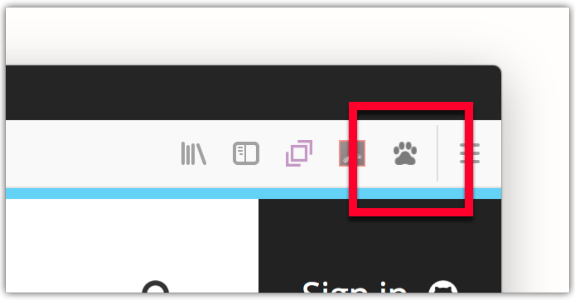


Figure 5.: A button added via the `browserAction` WebExtension API

It should be possible for custom qutebrowser extensions to add their own user interface elements. However, this functionality is unlikely to be needed when moving core functionality into extensions (which is the main focus of this SA). Therefore, this API is currently out of scope.

browserSettings

Enables an extension to modify certain global browser settings.

Extensions for qutebrowser should be able to modify its settings. Contrary to WebExtensions, all available settings can be exposed to extensions.

browsingData

Enables extensions to clear the data that is accumulated while the user is browsing.

No such functionality is currently implemented in qutebrowser, so this API is out of scope.

clipboard

The clipboard API enables an extension to copy items to the system clipboard.

Since qutebrowser extensions will have access to the Qt GUI library, no equivalent to this module is needed. Qt provides the `QClipboard`[14] class, which can be used for equivalent functionality.

[14] `http://doc.qt.io/qt-5/qclipboard.html`

commands

> Listen for the user executing commands that you have registered using the commands manifest.json key.

This API is used to register custom keyboard shortcuts in WebExtensions. A similar concept also exists in qutebrowser, where commands can either be bound to keys (using the configuration), or executed in the commandline via `:command`.

This API is important, since many core parts register commands in qutebrowser (like `:adblock-update` for the ad blocker). Thus, extensions should be able to register Python functions as commands.

contentScripts

> With the contentScripts API, an extension can register and unregister scripts at runtime.

Injecting custom JavaScript code into websites will be an useful feature for custom extensions at a later stage, but is not needed to move code out of qutebrowser's core, as it is expected for internal JavaScript code to stay in the core part. Thus, this module is currently out of scope.

Running JavaScript one-off snippets (triggered manually, rather than automatically on page load) is already implemented in the tab API and should be trivial to expose to extensions.

contextualIdentities, i18n, identity, idle, notifications, pkcs11, topSites

> contextualidentities: Work with contextual identities: list, create, remove, and update contextual identities.
>
> i18n: Functions to internationalize your extension. You can use these APIs to get localized strings from locale files packaged with your extension, find out the browser's current language, and find out the value of its Accept-Language header.
>
> identity: Use the identity API to get an OAuth2 authorization code or access token, which an extension can then use to access user data from a service which supports OAuth2 access (such as a Google or a Facebook account).
>
> idle: Find out when the user's system is idle, locked, or active.
>
> notifications: Display notifications to the user, using the underlying operating system's notification mechanism.
>
> pkcs11: The pkcs11 API enables an extension to enumerate PKCS#11 security modules, and to make them accessible to the browser as sources of keys and certificates.
>
> topSites: Use the topSites API to get an array containing pages that the user has visited often and frequently.

No such functionality is currently implemented in qutebrowser, so the above modules are out of scope.

cookies

> Enables extensions to get and set cookies, and be notified when they change.

Cookie access is not fully exposed by the QtWebEngine library[15], so it is currently impossible to implement this module.

devtools

> The devtools.inspectedWindow API lets a devtools extension interact with the window that the developer tools are attached to.
>
> The devtools.network API lets a devtools extension get information about network requests associated with the window that the devtools are attached to (the inspected window).
>
> The devtools.panels API lets a devtools extension define its user interface inside the devtools window.

Access to developer tools is not exposed by the QtWebEngine library, so it is currently impossible to implement this module.

dns

> Enables an extension to resolve domain names.

Since qutebrowser extensions will have access to the Qt GUI library, no equivalent to this module is needed. Qt provides the `QDnsLookup`[16] class, which can be used for equivalent functionality.

downloads

> Enables extensions to interact with the browser's download manager. You can use this API module to download files, cancel, pause, resume downloads, and show downloaded files in the file manager.

This module is needed for various core parts (such as the ad blocker, to download filter lists).

[15] `http://doc.qt.io/qt-5/qwebenginecookiestore.html`
[16] `http://doc.qt.io/qt-5/qdnslookup.html`

events, extensions, extensionTypes, permissions, runtime, types

> events: Common types used by APIs that dispatch events.
>
> extensions: Utilities related to your extension. Get URLs to resources packages with your extension, get the Window object for your extension's pages, get the values for various settings. Note that the messaging APIs in this module are deprecated in favor of the equivalent APIs in the runtime module.
>
> extensionTypes: Some common types used in other WebExtension APIs.
>
> permissions: Extensions need permissions to access more powerful WebExtension APIs. They can ask for permissions at install time, by including the permissions they need in the permissions manifest.json key. The main advantages of asking for permissions at install time are:
>
> runtime: This module provides information about your extension and the environment it's running in.
>
> types: Defines the BrowserSetting type, which is used to represent a browser setting.

The above APIs are specific to the WebExtensions API, and thus irrelevant for qutebrowser.

find

> Finds text in a web page, and highlights matches.

This functionality is already available as part of qutebrowser's tab API (see section 4.3), and should be trivial to expose to extensions.

history

> Use the history API to interact with the browser history.

Accessing and manipulating the history might be an useful feature for future user-contributed extensions, but is likely not needed for moving code out of qutebrowser's core. Thus, the history module is currently out of scope.

management

> Get information about installed add-ons.

Extensions are currently not expected to interact with each other (even less so when moving code out of the core), so this module is not needed.

omnibox

> Enables extensions to implement customised behavior when the user types into the browser's address bar.

When adding custom commands, extensions also should be able to specify a completion function, so users can use qutebrowser's autocompletion when typing the extension's commands.

privacy

> Access and modify various privacy-related browser settings.

In qutebrowser, these settings (like `privacy.network.webRTCIPHandlingPolicy`) are exposed as normal qutebrowser settings (like `content.webrtc_ip_handling_policy`), so there is no need for a similar module in the extension API.

proxy

> Use the proxy API to proxy web requests. There are two different ways you can do this:

No low-level networking access is possible via the QtWebEngine API, so this module will not be implemented.

search

> Retrieves search engines and executes a search with a specific search engine.

Search engines are part of the main configuration in qutebrowser, so an extension can trivially retrieve a search engine from there and execute a query without the need for a dedicated API.

sessions

> Use the sessions API to list, and restore, tabs and windows that have been closed while the browser has been running.

While qutebrowser does have a session feature, access to it is not vital for an extension API, so this is currently out of scope.

storage

> Enables extensions to store and retrieve data, and listen for changes to stored items.

Having a simple way to persist data for extensions will be an useful feature for third-party extensions, but is currently not needed when moving code out of the core. However, extensions should also be able to query the global data and configuration directories used.

tabs, windows

> tabs: Interact with the browser's tab system.
>
> windows: Interact with browser windows. You can use this API to get information about open windows and to open, modify, and close windows. You can also listen for window open, close, and activate events.

Various functionality already exists in qutebrowser as part of the "tab API" (documented in section 4.3), which should be trivial to expose to the extension API. Furthermore, the `tabs` and `windows` modules contains various functionality to retrieve tabs, which should be added to qutebrowser's extension API.

theme

> Enables browser extensions to update the browser theme.

No dedicated theme feature exists in qutebrowser – instead, qutebrowser's user interface can be customized using its configuration.

webNavigation

> Add event listeners for the various stages of a navigation. A navigation consists of a frame in the browser transitioning from one URL to another, usually (but not always) in response to a user action like clicking a link or entering a URL in the location bar.

This exposes various events such as `webNavigation.onCompleted`. These will be exposed as part of the tab API in qutebrowser, as Qt signals such as `load_finished`.

webRequest

> Add event listeners for the various stages of making an HTTP request. The event listener receives detailed information about the request, and can modify or cancel the request.

The `webRequest` module allows extensions to intercept and change HTTP network requests. Many of the fine-grained APIs available with WebExtensions (like `onResponseStarted` to modify the network response) are not available in the QtWebEngine API, so exposing them in qutebrowser's API is impossible. However, it is possible to intercept and block network requests, and such functionality is critical to move components like the ad blocker out from the core.

4.3. qutebrowser tab API

As explained in section 1.1, qutebrowser supports two rendering engines (QtWebKit and QtWebEngine) and provides an abstraction layer ("tab API") over those backends representing a single tab in the browser.

The tab API has been designed with clean API design in mind, as an extension API already was on the horizon when implementing it. Its goal is that the rest of qutebrowser's code never has to access backend-specific functionality (like the `QWebEngineView`[17] class) directly, and uses the abstraction layer instead.

The tab API is grouped into various classes:

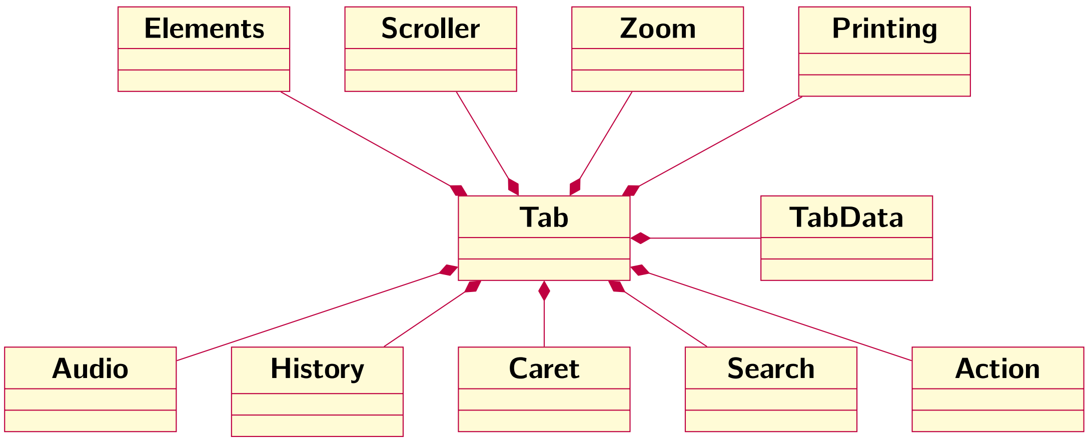


Figure 6.: Class diagram for existing tab API

With the exception of `TabData` which is backend-agnostic, all those objects exist as an abstract base class and as concrete implementations for QtWebKit/QtWebEngine each:

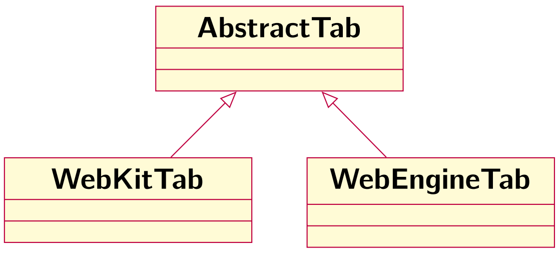


Figure 7.: Inheritance tree of tab API classes

The API exposed by those objects is quite long, so it is not included in full in this documentation.[18] As an example, the public API exposed in the main `Tab` object is listed on page 31.

Some parts of the API (like the `networkaccessmanager()` or `user_agent()` methods) are only exposed because there was a need in qutebrowser's core to do so, but should not be exposed via the extension API. However, for the most part, the tab API can be exposed unmodified to extensions and already allows for a wide range of interactions with tabs.

[17] `https://doc.qt.io/qt-5/qwebengineview.html`

[18] The full API is available at `https://github.com/qutebrowser/qutebrowser/blob/v1.5.2/qutebrowser/browser/browsertab.py`, accessed 2018-11-09.

```
window_close_requested = pyqtSignal()
link_hovered = pyqtSignal(str)
load_started = pyqtSignal()
load_progress = pyqtSignal(int)
load_finished = pyqtSignal(bool)
icon_changed = pyqtSignal(QIcon)
title_changed = pyqtSignal(str)
load_status_changed = pyqtSignal(str)
new_tab_requested = pyqtSignal(QUrl)
url_changed = pyqtSignal(QUrl)
shutting_down = pyqtSignal()
contents_size_changed = pyqtSignal(QSizeF)
add_history_item = pyqtSignal(QUrl, QUrl, str)
fullscreen_requested = pyqtSignal(bool)
renderer_process_terminated = pyqtSignal(TerminationStatus, int)
predicted_navigation = pyqtSignal(QUrl)

def event_target() -> QWidget
def progress() -> int
def load_status() -> LoadStatus
def title() -> str
def icon() -> QIcon
def networkaccessmanager() -> QNetworkAccessManager
def user_agent() -> str

def send_event(evt: QEvent) -> None
def handle_auto_insert_mode(ok: bool) -> None
def url(requested: bool) -> QUrl
def openurl(url: QUrl, predict: bool) -> None
def reload(force: bool) -> None
def stop() -> None
def clear_ssl_errors() -> None
def key_press(key: Qt.Key, modifier: Qt.KeyboardModifier) -> None
def dump_async(callback: Callable, plain: bool) -> None
def run_js_async(code: str, callback: Callable, world: JsWorld) -> None
def shutdown() -> None
def set_html(html: str, base_url: QUrl) -> None
```

Listing 1: Existing main tab API

4.4. Other inspirations

Various other existing projects served as an inspiration for qutebrowser's extension API:

4.4.1. pytest

The pytest project[19] exposes a very simple and "pythonic" extension API. All that is needed to create a plugin is a specially named Python file which implements hook functions such as `def pytest_runtest_setup(item)`. However, giving trainings about pytest in companies, the author of this SA has noticed that some aspects of its plugin API are somewhat unintuitive. As an example, both the function name and the argument names (`item`) need to match their definition; renaming the argument to `test_item` instead would lead to an error. While some of its design decisions make sense, others should be solved differently in qutebrowser.

4.4.2. odoo

The odoo project[20] also follows the philosophy of having a small core which is extended with modules (some of which are shipped with the core). From some prior work experience, this SA's author has worked with odoo modules in the past – unfortunately, the API for modules is complex and badly documented. Thus, odoo is mainly useful as a conceptual inspiration rather than an inspiration for API design.

[19]`https://www.pytest.org/`
[20]`https://www.odoo.com/`

5. Evaluation

5.1. Criteria

There are various forces which affect design decisions for qutebrowser's extension API:

Qt APIs qutebrowser is built on top of the QtWebEngine/QtWebKit rendering engines (which of the two to use is user-configurable). Sometimes, while a clean API design for a given problem would exist, constraints imposed by those libraries are a limiting factor and thus influence the API design. The API to get the selected text from a web page is an ideal example: The most straightforward API would be a `def selection() -> str` method. However, JavaScript execution is needed to get the selection, which is only available from Qt as a callback-based interface. Thus, the extension API will need to look like `def selection(cb: Callable[[str], None]) -> None` – in other words, the `selection` method will take a callback function, which then gets called with the selected text.

Internal qutebrowser code One of the main goals (as per the task description) of this SA is moving code from qutebrowser's core into internal "extensions" shipped alongside qutebrowser. Components which use general-purpose APIs (like the adblocker, which needs to intercept network requests) can conveniently moved out of the core, and result in extension APIs which are also usable for other purposes.

Ideas for future extensions While external extensions (contributed by the qutebrowser community) are not the primary focus of this SA, a lot of use-cases for extensions have been collected based on users' feature requests. Care should be taken so the extension API can also satisfy these use cases in the future.

Other extension APIs There is a general consensus from browser vendors around the WebExtension API. Unfortunately, that API is unsuitable for qutebrowser, for reasons explained in section 4.2. It can still serve as a source for inspiration.

Python and Qt While some higher-level architectural decisions are independent from the programming language used to implement them, what is commonly considered a "good" API certainly depends on the underlying programming language and the idioms used therein. Since qutebrowser's extension API is used from Python, it should aim to be "Pythonic" (i.e., adhering to Python idioms[1]) and also use features made available by Qt. As an example, a Pythonic API might favor Python decorators over inheritance to set up extension hooks; or a Qt API might prefer Qt's signals/slots facility callback-based API.

In addition to the mentioned external forces, it is helpful to have a set of self-imposed design guidelines in mind when designing an API.

[1] `https://blog.startifact.com/posts/older/what-is-pythonic.html`

For this project, it was decided to adopt the *"Characteristics of good APIs"* listed in Jasmin Blanchette's *Little Manual of API Design* (Blanchette 2008, 7ff):

Easy to learn and memorize: This also implies consistency and minimalism, as well as clear semantics, following the principle of least surprise.

Leads to readable code: As the manual puts it: *"Readable code can be concise or verbose. Either way, it is always at the right level of abstraction – neither hiding important things nor forcing the programmer to specify irrelevant information"*.

Hard to misuse: A well-designed API should assist its user in writing correct and clear code.

Easy to extend: New concepts will appear over time, and existing APIs will grow. While it is hard to anticipate future changes, they should be taken into account while designing an API.

Complete: The manual claims *"Ideally, an API should be complete and let users do everything they want."*. However, this statement is later revised by adding *"Completeness is also something that can appear over time, by incrementally adding functionality to existing APIs. However, it usually helps even in those cases to have a clear idea of the direction for future developments, so that each addition is a step in the right direction."*

5.2. Conclusion

Designing an API always is a trade-off between various, sometimes conflicting, forces. For qutebrowser's extension API, the focus should lie on being as minimal and straightforward as possible, with the primary goal of moving internal qutebrowser code to extensions.

While extensibility and completeness are valid concerns, they are less relevant as the API is not open to third-party extensions yet.

6. Concept

6.1. Summary

As shown in previous chapters, there are ample reasons for qutebrowser's extension API to differ from existing approaches in other browsers. Much of the functionality being used in extensions already exists as a cleanly defined API in qutebrowser, which could be exposed (after minor refactorings) in the extension API.

Due to time constraints, only small parts of the API were designed from scratch. Instead, the work being done was centered on improving and exposing APIs which already exist in qutebrowser.

6.2. Exposing existing APIs

Many of the APIs which should be exposed to extensions already exist in qutebrowser. However, exposing formerly private APIs carries certain risks:

- Often, APIs do not undergo any review before they are made public (Blanchette 2008, 18f), This can lead to mistakes which were not regarded as important (or not noticed at all) to be exposed publicly, without a way to remedy the mistakes (since that would be an incompatible change).
- Looking at the tab API explained in section 4.3, some methods exist because there was a need in the core part for them – but they should not be exposed to plugins, since they are only needed for very specialized reasons.
- When improving or refactoring APIs in the future, care must be taken because third-party extensions might be affected by the change. To avoid extension authors having to adjust their code with every upgrade of the core (and frustrating users due to breaking extensions), any future changes should be backwards-compatible, which greatly limits the room for future cleanups.
- Even for methods which seem useful for an extension API, their need should be carefully evaluated by looking at proposals for future extensions before blindly exposing them. This approach leads to a smaller (and thus simpler) API. It also increases the maintainability of the core, as parts which are not exposed can be freely changed without considering extensions.

It must be noted that many of these issues are less relevant in the scope of this SA, as the API will be open to third-party extensions at a later point. However, care was taken to mitigate them where possible.

In order to catch possible issues with existing APIs planning to be exposed to extensions, the existing APIs were reviewed by Raphael Das Gupta from the IFS (Institute for Software).

It was considered to implement the object adapter pattern (Johnson et al. 2005, p. 139) to limit the methods exposed in the API to what is actually necessary:

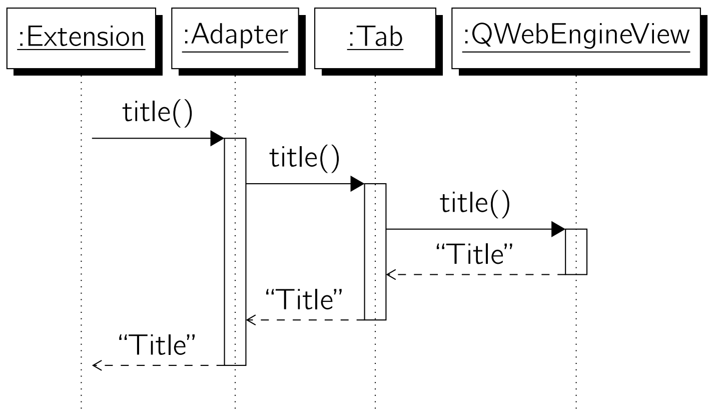


Figure 8.: Sequence diagram for the extension API with adapter object

However, this would lead to a considerable amount of duplicated code: Big parts of the tab API would need to be duplicated, just to pass calls through. Since this solution provided an overhead which was considered too big compared to the benefits, the following alternatives were considered:

- The methods could be marked private by following the common Python convention of starting the name with an underscore (for example, `_shutdown()`). However, in the core part, they should *not* be private, as they need to be used from outside the tab API.
- The methods could be lacking a Python docstring, so they are accessible, but excluded from the extension API documentation. However, auto-completion in development environments would still suggest those methods, which could lead to users thinking they are part of the exposed API. Additionally, this would make it impossible to document the affected methods for developers working on the core.
- Similar to existing members like `.scroller` or `.zoom`, a new `.private_api` member could be introduced, which points to an object containing any methods not part of the public API.

The last solution was chosen since it clearly communicates (both in the documentation and the name of the attribute) what part of the API is intended for usage in the core only. This solution also still allows the core to access those APIs, and the private API can still be documented properly.

Figure 9 show the class diagram for the refactored API. The full API definition can be found in the developer documentation in appendix B.

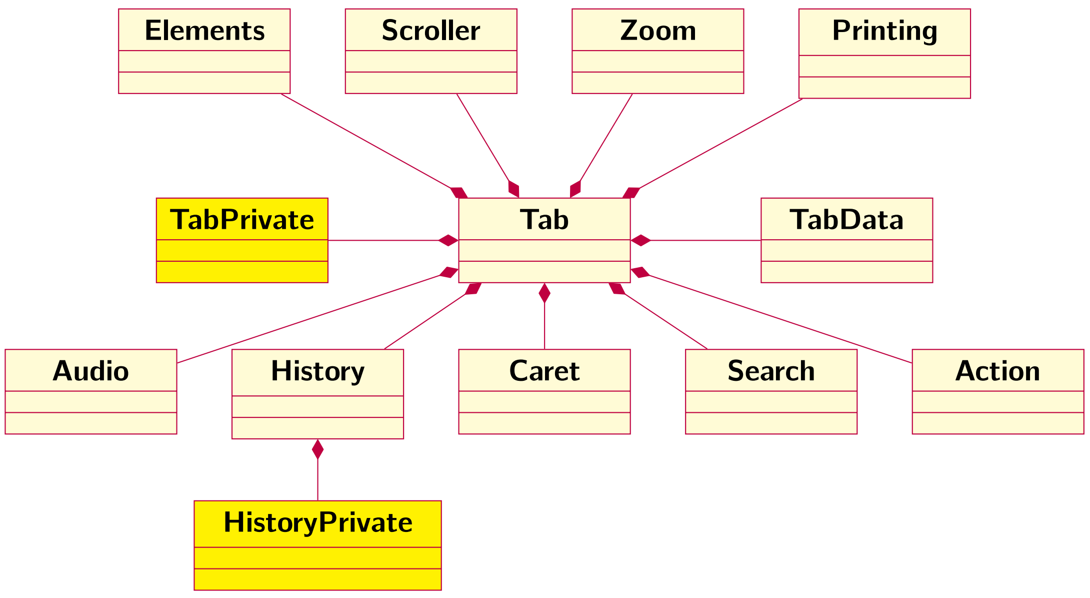


Figure 9.: Class diagram for refactored tab API, with added classes highlighted

6.3. Security

As part of this SA, third-party extensions are out of scope; only code which was part of qutebrowser's core is moved into extensions. Thus, it seems like there are no special security considerations to be made. However, the architecture of an extension API is fundamentally influenced by such considerations, and support for third-party extensions will be added in the near future. Therefore, the security philosophy of qutebrowser's extensions is analyzed in this section.

The security model of WebExtensions (see section 4.2) assumes that extensions are untrusted. Even with the limited WebExtensions API, malicious extensions are a common issue (Veditz 2015; Conca 2018). Browser vendors try to alleviate this problem with automated and manual code review, extension signing, and blacklisting of known-bad extensions. Additionally, the user explicitly needs to allow extensions to run in incognito/private browsing mode, as the impact of a privacy breach typically is considerably larger in that scenario.

The approach taken qutebrowser extensions is different: Extensions are treated as trusted, so users are responsible for reviewing extensions before installing them. This is for a variety of reasons:

- Extensions for qutebrowser should be written in Python (like qutebrowser itself is), but safely executing untrusted Python code is commonly regarded to be impossible (Batchelder 2012; Edge 2013).
- The attack surface for a malicious actor trying to distribute a bad extension is much smaller, since qutebrowser caters to a relatively small niche group of users. Thus, malicious extensions are expected to be a seldom occurrence compared to more common browsers such as Google Chrome or Mozilla Firefox.
- Users of qutebrowser are typically power users, due to its keyboard-focused nature. With a browser aimed at casual users, it might be easy to trick them into installing an extension

which is malicious. In contrast, users of qutebrowser can be expected to be much more diligent in ensuring that the extensions they are installing are not malicious.

- The volume of available extensions is much lower. Thus, it would be possible to create a whitelist of extensions which have been reviewed and approved by one of qutebrowser's core developers. A contributor wishing to distribute a new extension could then ask for it to be reviewed and included in that list.

- Due to the focus on power users, a user should always be able to install an extension manually, or write a custom one. Thus, mandatory extension signing or approval by a central body is undesirable, as it presents a trade-off between a user's freedom and security.

7. Implementation and Test

7.1. Type checking

7.1.1. Background

When it comes to type systems, programming languages are usually categorized using two properties: Being either *dynamically* or *statically* typed, and being either *weakly* or *strongly* typed.

Weak and strong typing describe how a language reacts when incompatible types are combined. Weakly typed languages (such as JavaScript) convert between types implicitly. In other words, in an expression such as `3 - '1'`, the string `'1'` is converted to a number (since it should be subtracted from another number), with the expression resulting in the number `2`.

Python is strongly typed, therefore, the same expression in Python raises an exception:

```
>>> 3 - '1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for -: 'int' and 'str'
```

Dynamic and static typing describe at what point the types need to be known. With statically typed languages, types need to be specified at compile time (such as `void func(int a)` in C). In more modern statically typed languages, types can often be inferred automatically instead of having to be specified by the programmer.

Python started as a dynamically typed language, which means that types are only known at runtime. However, with Python 3.0 (released in 2008), syntax for function annotations was added. Thus, the following code raised a `SyntaxError` in Python 2, but is valid syntax in Python 3:

```
def add(a: int, b: int) -> int:
    return a + b
```

However, how the type annotation syntax would be used was left to the applications using them. Annotations are completely ignored at runtime, which means a call like `add("1", "2")` would not be rejected based on annotations (and return the string `"12"`).

The annotation feature was not commonly used, but it started to gain traction in 2014 with PEP 484[1]. "Python Enhancement Proposals" (PEPs) are commonly written as a discussion ground for larger changes to the Python language or ecosystem.

PEP 484 was accepted in May 2015, and a `typing` module used to specify types was added to the Python 3.5 standard library, released in September 2015. The introduction of the `typing`

[1]https://www.python.org/dev/peps/pep-0484/

module allowed to specify types such as `Optional[int]` (either an integer, or the special `None` value) or `Mapping[str, int]` (a mapping from strings to integers).

However, Python remains a dynamically typed language, so annotations are still ignored at runtime. Apart from serving as documentation, there are various projects interpreting them:

- The PyCharm[2] IDE by JetBrains uses them for autocompletion and to notify the user about type errors.
- mypy[3] is the de-facto standard tool used to check type annotations for correctness: It can be run independently from Python and fails if there is a mismatch between the annotations and the implementation. Much of its development is supported by Dropbox.
- pytype[4] is an alternative to mypy, written by Google.
- pyre[5] is another alternative to mypy, written by Facebook.

Python's approach of not requiring type annotations, as well as delegating their interpretation to an external tool, is commonly referred to as *gradual typing*. A similar approach can be seen in Microsoft's TypeScript[6] language, which is type-checked and then compiled to JavaScript.

With both mypy and typescript, all type annotations are optional. Types which are not annotated are assumed to be of a special type `Any`. This type is compatible with any other type (thus, an object of the type `Any` can for example passed as an argument annotated with `int`), and also supports all operations. In other words, objects of type `Any` are not type-checked at all.

Additionally, by default mypy only type-checks the inside of functions which themselves have type annotations, in order to avoid flooding its user with false-positives.

This approach allows to incrementally introduce "static" typing into an existing codebase which previously had no type annotations at all, gradually introducing more type annotations over a longer time, fixing errors as they appear.

Since type checking in Python is a relatively new feature, not all third-party libraries used by qutebrowser come with type information. For Python's standard library and some third-party libraries, the typeshed[7] project exists, where type information is maintained separately from Python itself.

Additionally, a more recent change to the Python ecosystem (PEP 561[8], accepted in June 2018) also allows type information to be distributed independently from a library package itself. A PEP-561 package consists of so-called "stub files" with a `.pyi` extension, similar to `.h` header files in C or C++.

[2] https://www.jetbrains.com/pycharm/
[3] http://www.mypy-lang.org/
[4] https://github.com/google/pytype
[5] https://pyre-check.org/
[6] http://www.typescriptlang.org/
[7] https://github.com/python/typeshed
[8] https://www.python.org/dev/peps/pep-0561/

7.1.2. Introducing type checking

As per the task description, type annotations were introduced into qutebrowser's code base. Before starting this project, very few functions already carried type annotations (when contributed by a developer who prefers using annotations), but the majority of qutebrowser's code was unannotated. Additionally, no type checker was run systematically as part of qutebrowser's continuous integration toolchain.

The tools mentioned in 7.1.1 were evaluated, with the mypy tool being selected. It is widely used and backed by a more diverse community than either pytype and pyre (which are mostly developed and used by Google and Facebook, respectively).

The introduction of mypy was split into several phases, with the goal of getting mypy to run without any errors/warnings as quickly as possible, and then making its checking stricter or more accurate.

Initial run

When running mypy over qutebrowser's codebase without any changes, 324 errors were reported. Most errors looked similar to:

```
qutebrowser/utils/qtutils.py:36: error:
No library stub file for module 'PyQt5.QtCore'
```

with mypy complaining that it found no type information for various PyQt modules.

Ignoring missing imports

Mypy provides a `--ignore-missing-imports` switch which allows ignoring such missing modules. Note that its usage is discouraged in mypy's documentation: *"We recommend using this approach only as a last resort"*[9]. Nevertheless, it was deemed useful in order to focus on other error messages first.

With the `--ignore-missing-imports` option, mypy only produced 88 errors.

Analyzing these errors revealed that they could be split into various categories:

- Mypy not recognizing how qutebrowser adds an additional `VDEBUG` (verbose debugging) logging level to Python's logging system. This was solved by adding *`# type: ignore`* comments to make mypy ignore those issues as appropriate.

- Trouble with a common pattern of handling optional library imports like so:

```
try:
    import secrets
except ImportError:
    secrets = None
```

[9]`https://mypy.readthedocs.io/en/latest/running_mypy.html#missing-imports`, accessed 2018-12-03

There, mypy complained since setting a module type to the `None` value is an invalid operation from the type system's view, but it still perfectly valid (if later guarded via `if secrets is not None:`) in Python. Thus, these issues also were ignored by adding `# type: ignore` comments. An issue for better handling of this case in mypy was already open in mypy's issue tracker: `https://github.com/python/mypy/issues/1153`.

- Mypy not handling Python's `enum.IntEnum` correctly when used via its function-based (rather than class-based) API. This was worked around by using the class-based API instead; an issue was already open in mypy's issue tracker: `https://github.com/python/mypy/issues/4865`.
- Additional type annotations being needed by mypy in places where types could not be inferred (such as `cmd_dict = {}  # type: typing.Dict[str, Command]`).
- Module-level globals which are set to `None` initially and then initialized in an `init()` method. Mypy asserted that those could be `None` throughout the codebase (thus resulting in many potentially invalid operations), while their initialization happens very early, so they can be safely assumed to never be `None` despite their initial value. This was solved by overriding the assumed type via `instance = typing.cast('Config', None)`.
- Other values which similarly never are `None` when a certain function gets called, despite them having been `None` at some point before. Since mypy understands additional hints in the form of `assert val is not None` or `assert isinstance(index, int)`, such assertions could be used to silence these errors.
- One case were an existing `typing.Union[str, int]` annotation falsely claimed that a value could be either `str` or `int`, while in reality it always was a string.
- Various other cases where existing type annotations had to be refined to be more accurate.

After fixing these issues, mypy finished without showing any errors, but without having type information for libraries being used by qutebrowser.

Adding library type information

After removing the `--ignore-missing-imports` switch again, the main issue was missing stubs for PyQt, the main library being used by qutebrowser. While PyQt comes with a way to generate stub files from C++ type information (being a quite simple wrapper over C++ code), those stubs come with various issues making them unsuitable for use with mypy.

As a substitute, a PyQt5-stubs project exists[10], packaging adjusted type stubs for PyQt5 as a separate package (via PEP 561, see section 7.1.1). With those installed and the switch removed, 55 new errors appeared. These again could be classified into various categories:

- Issues with the PyQt5-stubs project where type annotations were inaccurate. Those were fixed in a copy (GitHub fork) of the stubs in the qutebrowser organization[11] and contributed to the upstream project. However, a reply from its maintainer is still pending as of December 2018, so qutebrowser currently installs its own copy instead.

[10] `https://github.com/stlehmann/PyQt5-stubs`
[11] `https://github.com/qutebrowser/PyQt5-stubs`

- Missing stub files for Python standard library modules in the typeshed project. For the `faulthandler` module, writing stubs was straightforward, so they were contributed[12] to the typeshed project. For more complex cases, modules were ignored in the mypy config file instead.
- Bugs in the type annotations in typeshed, for which fixes were contributed[13] as well.
- Missing type annotations for other third-party modules. For these, an issue was opened in order to inform the respective developers about the possibilities to add type information to their packages, and the module was then ignored via the mypy config file.
- New places where mypy needed some additional type annotations in qutebrowser's code for types it could not infer correctly.

7.1.3. Increasing strictness

After returning to a clean mypy invocation with additional type information, it was attempted to add the `--strict` switch to mypy, causing it to be more pedantic about various checks. However, after enabling the strict option, mypy displayed 5385 new errors. Most new errors were due to untyped functions (and calls into such functions) being prohibited entirely in that case.

Instead, the more fine-grained options implied by `--strict` were evaluated individually:

–warn-unused-configs warns about unused module-specific settings in the config file, which could be caused by a typo. However, this could not be turned on due to a bug[14] causing false-positive warnings in mypy.

–disallow-subclassing-any disallows subclassing from an object of an unknown (`Any`) type. This was enabled globally except for a module (`browser.webkit.rfc6266`) in qutebrowser describing a parser grammar, which relies on dynamic typing.

–disallow-untyped-calls disallows calls into any function which does not have type information. This was causing a huge number of errors (since most of qutebrowser's codebase is not annotated yet), and thus was not enabled.

–disallow-untyped-defs disallows defining any function without type annotation. This was not enabled since qutebrowser's codebase will gradually become more annotated. However, it was enabled for modules which were annotated as a part of this SA, so that any future additions to those modules will also need to carry type information.

–disallow-incomplete-defs disallows partially annotated functions, where only some arguments are annotated. This was initially enabled and some partial annotations were completed, but it was later disabled due to false-positives in mypy[15].

–check-untyped-defs causes mypy to check the inside of functions which are not annotated yet. This caused 392 new errors and thus could not be enabled yet. However, reviewing these

[12] `https://github.com/python/typeshed/pull/2627`
[13] `https://github.com/python/typeshed/pull/2635`, `https://github.com/python/typeshed/pull/2636`
[14] `https://github.com/python/mypy/issues/5957`
[15] `https://github.com/python/mypy/issues/5954`

errors revealed that many were false-positives or mypy limitations, but also uncovered a crash (which was then fixed) in qutebrowser.

–disallow-untyped-decorators disallows applying a decorator to a function which strips its type information (since the type of the decorator is unknown). This was enabled as it did not lead to any new errors.

–no-implicit-optional disallows annotations with a `None` default value. An example of such an annotation is `def f(a: str = None)`. Instead, this option requires the more accurate `typing.Optional[str]` annotation. Since this would lead to more verbose type annotations without any perceived benefit (the same information can be seen by looking at the default value), it was not enabled.

–warn-redundant-casts warns about constructs like `typing.cast(int, x)` when `x` already is of type `int`. It was enabled since it did not result in any new errors and such casts are likely to be unintended.

–warn-unused-ignores warns about `# type: ignore` comment on lines where no error would have had occurred. This warning can be useful when such comments are added because of a bug in mypy or type definitions, so it can be removed when the bug was fixed upstream. This was enabled as it did not result in any new errors.

–warn-return-any warns when a function returns a value of unknown (`Any`) type, since that can propagate into other places which then are not type-checked either. This was not enabled as doing so would require many additional type annotations for qutebrowser's codebase.

7.1.4. Adding annotations

After mypy's configuration was completed and it still ran without any issues, additional type annotations were introduced in qutebrowser's codebase. The task description requires anything exposed by the extension API to carry type annotations. Additionally, much of the config system (which is not exposed yet) was also annotated in order to gain some additional experience with mypy.

This resulted in the following modules being fully annotated (in addition to some being partially annotated):

- `browser.browsertab` ("tab API")
- `browser.webelem`, `browser.webkit.webkitelem`, `browser.webengine.webenginelem` ("web element API")
- `misc.objects` (various application objects)
- `commands.cmdutils` (registering commands)
- All modules in the `config` package: `config`, `configcache`, `configcommands`, `configdata`, `configdiff`, `configexc`, `configfiles`, `configinit`, `configtypes`, `configutils`, `websettings`.
- All modules in the `api`, `components` and `extensions` sub-packages (see section 7.2).

7.2. Important packages, modules and classes

The modules added for qutebrowser's extension API are spread across three packages:

- The `qutebrowser.api` package contains the API exposed to extensions (in other words, any code which is part of the extension API).
- The `qutebrowser.extensions` package contains supporting infrastructure and internal code for handling extensions.
- The `qutebrowser.components` package contains modules which formerly were part of qutebrowser's core and only use the extension API – that is, they only import code from the `qutebrowser.api` package, but not from any other `qutebrowser.*` packages.

7.2.1. The qutebrowser.api package: Public extension API

Module	Description
`apitypes.py`	Various basic types which can be used by extensions. These are either used as type annotations (such as `Tab`) or as enumerations (such as `ClickTarget` which is used to specify how to open a clicked link).
`cmdutils.py`	Utilities related to registering command handlers from extensions, such as the `@cmdutils.register()` decorator.
`config.py`	Access to the config from extensions, by either using a shorthand like `config.val.content.javascript.enabled`, or the `get()` function like `config.get('content.javascript.enabled')`.
`downloads.py`	Used to trigger download of temporary files (such as adblock filter lists) from extensions. In the future, this module could be extended to allow interacting with existing downloads triggered by the user.
`hook.py`	Allows extensions to register hooks for certain events via decorators, such as `@hook.init()` or `@hook.config_changed()`.
`interceptor.py`	Can by used by extensions to register a *request interceptor*, which then gets called for every network request made by qutebrowser. Based on the URL of the page and the URL being requested, the extension may decide to block the request.
`message.py`	Used to show messages to the user via functions like `message.info("...")` or `message.error("...")`.

Table 7.1.: Modules in the qutebrowser.api package

The `qutebrowser.api` package is described in more detail in the developer documentation in appendix B.

7.2.2. The qutebrowser.extensions package: Internal extension machinery

The `qutebrowser.extensions` package consists of the following two modules: explained in further detail below.

extensions.interceptors module

This module implements the internal logic so extensions can intercept and optionally block network requests made by qutebrowser.

The classes and functions which are part of this module are listed in table 7.2.

extensions.loader module

This module implements the internal loading and initializing of extensions/components. It is responsible for dynamically finding all modules in the `qutebrowser.components` package, loading them, and calling their registered hooks correctly.

The classes and functions which are part of this module are listed in table 7.3.

7.2.3. The qutebrowser.components package: Code moved out of the core

The `qutebrowser.components` package contains code which was moved out of qutebrowser's core, and now only accesses the `qutebrowser.api` package, without interacting with any other parts of qutebrowser's code. It contains the following modules:

Module	Description
`adblock.py`	qutebrowser's adblocker implementation, blocking advertisements in websites.
`caretcommands.py`	Commands related to moving the caret (cursor) around via keybindings, such as `:move-to-end-of-document` or `:toggle-selection` (18 commands total).
`misccommands.py`	Miscellaneous qutebrowser commands, such as `:home`, `:reload` or `:print` (15 commands total).
`scrollcommands.py`	Commands related to scrolling (`:scroll`, `:scroll-px`, `:scroll-to-perc`, `:scroll-to-anchor`).
`zoomcommands.py`	Commands related to zooming (`:zoom-in`, `:zoom-out`, `:zoom`).

Table 7.4.: Modules in the qutebrowser.components package.

Class / Function	Description
`Request`	A class representing a network request, containing information such as `first_party_url` (the page being visited) or `request_url` (the URL of the resource being requested). The request can be blocked by calling its `block()` method.
`InterceptorType`	The type of an interceptor function, intended to be used in type annotations (exposed to extensions as `qutebrowser.api.interceptor.InterceptorType`).
`register()`	Register a new request interceptor (exposed to extensions as `qutebrowser.api.interceptor.register()`).
`run()`	Used internally by qutebrowser to run all registered interceptors.

Table 7.2.: Important classes and functions in the qutebrowser.extensions.interceptors package

Class / Function	Description
`InitContext`	Information passed to an extension when it gets initialized (if it declares a function decorated with `@hook.init()`, see `hook.py` in table 7.1). Contains information such as the commandline arguments passed to qutebrowser, or the data/config directories used. Used via a `_get_init_context()` factory method.
`ModuleInfo`	Information attached to a Python module object. It is used to record internal information by decorators like `@hook.init()` (see `hook.py` in table 7.1).
`ExtensionInfo`	Meta-information about an extension. Currently only contains the name of an extension, but could be extended in the future to record additional information such as version numbers or the author of a third-party extension.
`add_module_info()`	Used internally to add a `ModuleInfo` instance to a Python module object.
`load_components()`	Finds and loads all modules in the `qutebrowser.components` package (see section 7.2.3). Uses a `_load_component()` utility function internally which loads a single component.
`walk_components()`	Find all available components. Used by `load_components()` and by qutebrowser's packaging infrastructure so all component modules are added to Windows/macOS builds). Uses two different implementations internally (`_walk_normal()` and `_walk_pyinstaller()`) because the simpler approach does not work in builds built via PyInstaller.
`_on_config_changed()`	Triggered on a configuration change, takes care of finding and calling all extension methods decorated with `@hook.config_changed()` (see `hook.py` in table 7.1).

Table 7.3.: Important classes and functions in the qutebrowser.extensions.loader package

7.3. Automated Testing

When this project was started, the qutebrowser project already existed – it was started in December 2013. Thus, a comprehensive test suite was already present at the time:

- Around 7400 tests (including parametrizations with different data)
- Including fuzzing (via hypothesis[16]) and benchmark tests (via pytest-benchmark[17])
- Testing on Linux, macOS and Windows
- Testing with all supported Python versions (3.5, 3.6 and 3.7)
- Testing with all supported Qt versions (5.7, 5.9, 5.10, 5.11, 5.12)
- Average test coverage of 80% (including branch coverage)

Therefore, the aim in this project was to build upon the existing testing infrastructure, and make sure the total coverage does not decrease with the newly added code. Ideally, the coverage of newly added modules should surpass the existing average coverage of 80%[18].

Unfortunately, there were no good uses for the hypothesis (fuzzing, i.e., testing with many random values) and pytest-benchmark libraries while adding tests for the code added/changed during this project. The test coverage for modules added for the extension API (or modules with major changes) is shown in table 8.2.

Based on this data, the goal of surpassing the existing average coverage of 80% was met: In the files relevant to this SA, an average test coverage of 90% has been achieved.

A test output log can be found in appendix E.

7.4. Manual Testing

The following tests were (successfully) performed manually, since related automated tests were insufficient:

- Updating adblock filter lists via the `:adblock-update` command; verifying that lists are downloaded correctly and a `adblock: Read ... hosts from 1 sources.` message is shown.
- Verifying that visiting a website containing ads (`https://www.20min.ch/`) is displayed without ads.
- Running `:print` to display the print dialog and printing the website into a PDF file; using the `:print --pdf` file to generate a PDF file directly.
- Verifying that the contents of those PDFs look as expected.

[16]https://hypothesis.works/
[17]https://pytest-benchmark.readthedocs.io/
[18]The author is well aware that coverage alone is not necessarily a useful metric. Blindly following a "coverage goal" thus is of little value. However, it is useful when using it as a guideline while consciously developing good tests, which is what has been done.

Directory	File	Coverage (%)	Comment
api/	*Average*	80	
	apitypes.py	100	
	cmdutils.py	100	
	config.py	100	
	downloads.py	70	Requires a download manager object for which no stub/mock exists yet.
	hook.py	90	
	interceptor.py	100	
	downloads.py	100	
components/	*Average*	85	
	adblock.py	93	
	caretcommands.py	82	
	misccommands.py	73	Some commands (like printing) contain hard to test GUI-interactions.
	scrollcommands.py	96	
	zoomcommands.py	85	
extensions/	*Average*	89	
	interceptors.py	94	
	loader.py	88	
browser/	browsertab.py	87	
commands/	command.py	89	
Average		90	

Table 7.5.: Test coverage for added/changed modules

8. Results

8.1. Achievement of Objectives

In this chapter, the objectives set in section 1.3 will be reviewed.

8.1.1. Merging contributions

At the beginning of this project, various third-party contributions were pending a code review, due to being ignored during the exam session preceding this semester.

Since those contributions would conflict with the refactorings needed before work on the extension API started, a block of time was reserved for code reviews at the beginning of this project (see figure 3 on page 16).

To avoid receiving more contributions while refactoring the code, the following notice was added to qutebrowser's contribution guidelines:

> Important: *Currently, bigger changes are going on in qutebrowser, as part of a student research project about adding a plugin API to qutebrowser and moving a lot of code from the code into plugins.* Due to that, bandwidth for pull request review is currently very limited, and contributions might lead to merge conflicts due to ongoing refactorings.

The impact of such a notice is hard to measure, but pull requests (where contributors request their changes to be pulled into the main repository) continued to be opened regularly throughout the project. Thus, the decision was taken to continue with the next step despite many remaining open contributions, and only merge new contributions if they are trivial enough.

8.1.2. Refactorings

In section 1.3.2, three major refactorings related to the extension API were identified:

- qutebrowser should introduce "gradual typing" and a type checker such as mypy into its toolchain. This goal was fully achieved despite some issues with third-party projects (which were fixed as part of this project), see section 7.1.
- The documentation toolchain used in qutebrowser should be switched to the Sphinx tool. Some work on this was started by an external contributor, but not finished in time, which was identified as one of the possible risks in section 2.5. To remedy this, Sphinx was introduced alongside the existing documentation toolchain, and only used to document the extension API (see appendix B). The author of this report (Florian Bruhin) will meet with

the external contributor (Fritz Reichwald) at the 35th Chaos Communication Congress[1] after this project is done (December 27th to 30th, 2018). There, they plan to finish migrating the entire existing documentation to the Sphinx tool.

- The "object registry" (`objreg` module) in qutebrowser should be refactored, as it has various issues and influences the public API. Work on this goal was started but not completed due to some deeper issues with its implementation[2]. Fixing these issues properly would take more time than anticipated, thus this refactoring was not completed. However, care was taken to not add the object registry to the API exposed to extensions, so this change can still be done after the API has been implemented.

The initial refactoring goals were only met partially – however, it was possible to continue with implementing the extension API despite that. Care was taken to make sure that work absolutely necessary for the extension API was finished.

8.1.3. Extension API

While the extension API resulting from this work is rather minimal, it is already quite powerful: As described in section 7.2.3, it allowed various components to be moved from qutebrowser's core to use the extension API. It also implements dynamic loading of extensions, as a first stepping stone towards loading third-party extensions in the future.

Originally, it was intended to start opening the API to third-party extensions as part of this project – however, this decision was later revised: Since the API is very new, it should be allowed to gain some maturity before making it public. Otherwise, problems noticed in the API could not be fixed easily, as doing so would result in a breaking API change.

8.1.4. Documentation

The value of a good documentation for work like this is not to be underestimated. However, neither is the effort that needs to be put into such a documentation, especially since both this documentation and the resulting code were written by a single author. Juggling both code and documentation simultaneously was difficult at times, but their author is pleased with the outcome of both. This project also showed that documenting thoughts and ideas lead to a clearer picture of how the resulting code should look.

8.2. Releases

In section 2.1, various qutebrowser releases were scheduled. v1.5.0 was released in week 3, following the release of PyQt 5.11.3 as planned. Patch release v1.5.1 was shipped in week 4; v1.5.2 in week 6. No further patch releases were needed. The release of v1.6.0 was originally intended for week 12, however, it was decided to delay it to January. This allows synchronizing it with the release of PyQt 5.12[3].

[1] `https://en.wikipedia.org/wiki/Chaos_Communication_Congress`

[2] See `https://github.com/qutebrowser/qutebrowser/issues/640#issuecomment-443143463`

[3] `https://www.riverbankcomputing.com/pipermail/pyqt/2018-December/041192.html`

8.3. Risk review

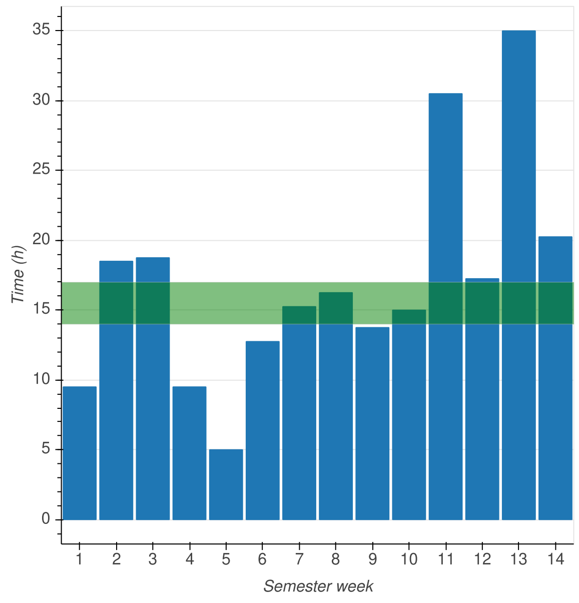
At the beginning of this project, various possible risks with this project were analyzed (see section 2.5). This section reviews the evaluated risks at the end of this project.

- Too many external contributions: Like predicted, third-party contributions continued to be submitted (see section 8.1.1). After it was clear that reviewing all of them in time would not be possible, work on the extension API continued with new contributions being ignored until after this work was complete.
- Little time in construction phase: Since reviewing existing contributions took a long time, only little time was available for construction. Despite that, it was still possible to move various qutebrowser commands and a major component (the adblocker) out of the core.
- Sickness of author: The author of this document caught a fever in week 5. Thankfully, it only lasted four days, and thus did not have a major impact.
- Migrating documentation toolchain is not done in time: This was indeed the case, but it was possible to generate API documentation with the Sphinx tool independently from the existing qutebrowser documentation.

A risk which was not considered is hardware failure – the author's laptop refused to turn on in week 13, shortly before this project was due. Fortunately, finding and pressing a hidden reset button brought it back to life, but about half a day of time was lost due to the incident.

8.4. Time evaluation

Based on the 30 ECTS points awarded for the SA, a workload of 14–17 hours per week is expected (see section 2.4). Analyzing the spent time each week shows the following distribution:



Week	Time (h)	Week	Time (h)
1	10	8	16
2	18	9	14
3	19	10	15
4	10	11	30
5	5	12	17
6	13	13	35
7	15	14	20

Total: 237 hours

Figure 10.: Time spent per week

The time entries tracked were split into various categories:

Category	Description	Time (h)
Documentation	Writing this documentation and working on API documentation.	88
Implementation	Implementing the extension API, introducing mypy, fixing bugs.	60
Contributions	Reviewing pull requests and contributions.	34
Meetings	Meetings with Prof. Stefan Keller, language reviews with AnneMarie O'Neill, video calls with Fritz Reichwald (see section 2.2).	29
Other	Reading API design literature, communication with community, fixing issues with upstream library upgrades, etc.	27

Table 8.1.: Time tracking categories

Reviewing the time tracked per week per category, the project phases initially defined in the schedule on page 16 are clearly apparent:

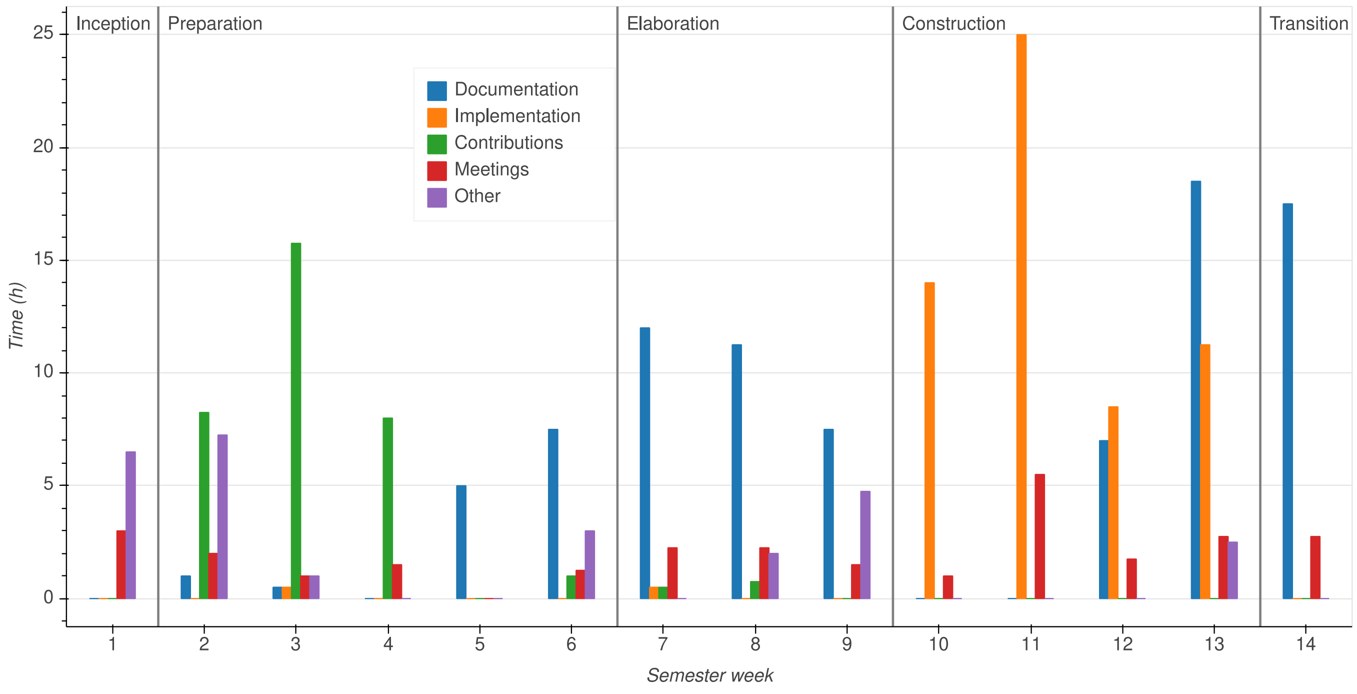


Figure 11.: Time tracking categories per week

Time was tracked using Clockify[4]. Analysis and visualization of the resulting data was done via a custom Python script and the Bokeh[5] library.

[4]https://www.clockify.me/
[5]https://bokeh.pydata.org/

8.5. Code Statistics

Since the work on this project was on an existing codebase, it is difficult to distinguish in code statistics what work was pre-existing, compared to work done as part of the project.

In the following table, code statistics as calculated by the cloc[6] tool ("count lines of code") are presented for all files created as part of this project, as well as files with major changes:

Directory	File	Blank	Comment	Code	Func./Meth.	Classes
api/	*Total*	133	304	111	20	6
	__init__.py	4	22	0	0	0
	apitypes.py	3	19	5	0	0
	cmdutils.py	59	108	52	6	3
	config.py	8	29	6	1	0
	downloads.py	20	33	22	3	1
	hook.py	26	46	20	8	2
	interceptor.py	10	28	5	2	0
	message.py	3	19	1	0	0
components/	*Total*	213	338	556	65	2
	__init__.py	2	18	0	0	0
	adblock.py	64	76	207	19	2
	caretcommands.py	52	72	87	18	0
	misccommands.py	61	91	160	21	0
	scrollcommands.py	20	44	58	4	0
	zoomcommands.py	14	37	44	3	0
extensions/	*Total*	70	68	112	13	4
	interceptors.py	20	24	19	3	1
	loader.py	50	44	93	10	3
browser/	browsertab.py	256	251	617	136	15
commands/	command.py	75	130	371	20	2

Table 8.2.: Line count for added/changed modules

Notes:

- `browser/browsertab.py` is relatively large for a single file. If any future additions are required, it should be split into smaller files.
- `__init__.py` files mark a folder as a Python package, thus do not contain any code.
- Some files in `api/` only import code from internal qutebrowser modules in order to expose them to the extension API, and thus do not contain any functions/classes on their own.
- Documentation (such as the API documentation in appendix B or the user documentation for qutebrowser commands) is generated from Python docstrings in the code, which cloc counts as "comments".

[6]`https://github.com/AlDanial/cloc`

8.6. Demo extension

In this section, a demo extension using various aspects of the API is presented. The extension checks whether a website uses the Leaflet[7] JavaScript library, which is often used to embed OpenStreetMap[8] maps into websites.

First, the extension imports the `typing` module from Python's standard library, and various submodules of the `qutebrowser.api` module:

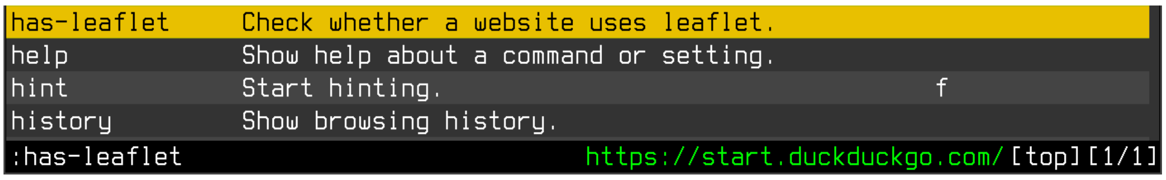
```python
import typing

from qutebrowser.api import cmdutils, apitypes, hook, message
```

Next, it registers a command handler using the `@cmdutils.register()` decorator, which causes the function to be available to the user as a `:has-leaflet` command inside qutebrowser:

```python
@cmdutils.register()
@cmdutils.argument('tab', value=cmdutils.Value.cur_tab)
def has_leaflet(tab: apitypes.Tab) -> None:
    """Check whether a website uses leaflet."""
```

The command name is automatically deduced from the name of the function, while the Python docstring gets shown in qutebrowser's command completion:



```
has-leaflet         Check whether a website uses leaflet.
help                Show help about a command or setting.
hint                Start hinting.                                f
history             Show browsing history.
:has-leaflet                         https://start.duckduckgo.com/ [top] [1/1]
```

Figure 12.: Command completion showing the `:has-leaflet` command

The `@cmdutils.argument(...)` decorator is used to pass more information about the `tab` argument to the API. Due to the `value=cmdutils.Value.cur_tab` parameter, the `tab` argument is set to the tab object of the currently focused tab when qutebrowser calls the command handler.

Next, the extension uses the tab API to find HTML elements on the website:

```python
    tab.elements.find_css('.leaflet-container',
                          callback=show_message,
                          error_cb=show_error_message)
```

The `find_css()` method takes three arguments: A CSS selector, a callback which gets called on success, and a callback which gets called in case of an error.

The CSS class selector `.leaflet-container` is used to find websites using Leaflet, as the library uses that selector for its map display.

[7] https://leafletjs.com/
[8] https://www.openstreetmap.org/

The callback used for errors simply uses the `message` module to show an error message to the user:

```python
def show_error_message(error: Exception) -> None:
    message.error(str(error))
```

The callback called on success instead checks whether the `elements` lists it received contains any elements. Depending on the result, it shows the appropriate message:

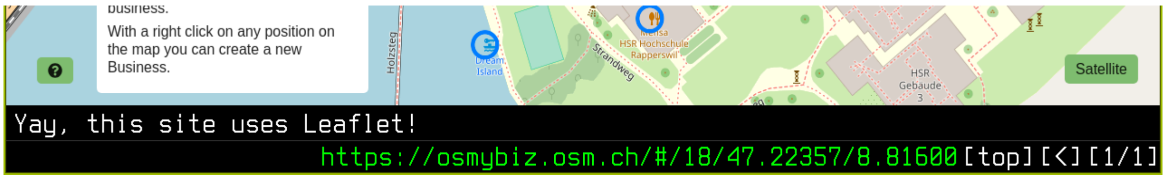
```python
def show_message(elements: typing.List[apitypes.WebElement]) -> None:
    if elements:
        message.info("Yay, this site uses Leaflet!")
    else:
        message.info("This site does not use Leaflet...")
```

Finally, the extension uses the `@hook.init()` decorator to define an initialization hook. There, it displays an "has-leaftlet initialized" message:

```python
@hook.init()
def init(_context: apitypes.InitContext) -> None:
    message.info("has-leaftlet initialized")
```

Note that it receives an `InitContext` object with additional information (such as commandline arguments). Since the argument is unused, its name is prefixed by an underscore, a common convention in Python code.

The full code of the demo extension can be found on page 57.


business.
With a right click on any position on the map you can create a new Business.

```
Yay, this site uses Leaflet!
                  https://osmybiz.osm.ch/#/18/47.22357/8.81600[top][<][1/1]
```

Figure 13.: Running the `:has-leaflet` command on OSMyBiz

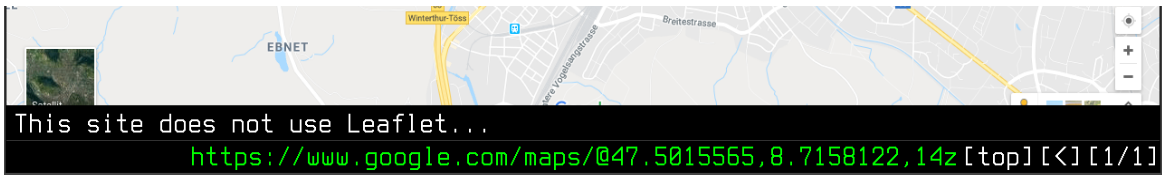

```
This site does not use Leaflet...
          https://www.google.com/maps/@47.5015565,8.7158122,14z[top][<][1/1]
```

Figure 14.: Running the `:has-leaflet` command on Google Maps

```python
import typing

from qutebrowser.api import cmdutils, apitypes, hook, message


@cmdutils.register()
@cmdutils.argument('tab', value=cmdutils.Value.cur_tab)
def has_leaflet(tab: apitypes.Tab) -> None:
    """Check whether a website uses leaflet."""
    tab.elements.find_css('.leaflet-container',
                          callback=show_message,
                          error_cb=show_error_message)


def show_error_message(error: Exception) -> None:
    message.error(str(error))


def show_message(elements: typing.List[apitypes.WebElement]) -> None:
    if elements:
        message.info("Yay, this site uses Leaflet!")
    else:
        message.info("This site does not use Leaflet...")


@hook.init()
def init(_context: apitypes.InitContext) -> None:
    message.info("has-leaftlet initialized")
```

Listing 2: Demo extension

8.7. Future Work

While there was a lot of progress for extensions in qutebrowser, there is plenty of room for future research and improvements:

- The core of qutebrowser should be further modularized. Ideally, its architecture would be designed with the microkernel pattern (Buschmann et al. 1996, 171ff) for operating systems in mind, however, the resulting extension API should always be general enough to be useful for more than one use-case.
- After the extension API is stable enough, extensions should be made open for third-party contributions, initially by allowing users to download extensions they want to use in a folder such as `~/.local/share/qutebrowser/extensions`.
- Many use-cases and ideas for extensions have been suggested[9] by qutebrowser's users. These should be carefully reviewed and considered for future API additions.
- In order to foster community involvement around extensions and keep users safe, a central place to distribute and update extensions should exist. Furthermore, an "extension store" should be added to qutebrowser, which allows users to easily discover, install and upgrade extensions.

The currently existing API has been designed with future additions in mind: The `downloads.py` module currently only allows triggering a temporary download, but was split off from other modules in order to allow for future additions for handling user-initiated downloads. Furthermore, init-hooks which can be defined by extensions receive a special `InitContext` instance (rather than getting called with the information therein as individual arguments) so that the `InitContext` class can be extended in the future without extensions needing to adapt their code.

8.8. Personal Review

Exactly one week before this project was due, December 14th 2018, qutebrowser celebrated its fifth birthday. Back then, I was not sure whether I wanted to pursue a Bachelor's degree at all, or whether I would rather work in the industry immediately. Later, when it was clear that I wanted to study at HSR, I wondered whether I would ever have the chance to work on qutebrowser as part of my studies.

I was happy that this was indeed the case, thanks to the flexibility of the HSR and Prof. Stefan Keller. There are a lot of other interesting research project proposals, so I am honored that this arrangement was possible.

Working on an existing project proved to be challenging, as it was a "moving target" at times with many external contributors involved. Even though I planned a phase at the beginning for reviewing existing contributions, it became apparent that reviews would take much longer. Additionally, contributions continued despite a notice asking people to hold them back if possible. In retrospective, a better idea would be to put up such a notice very early, shorten the time for

[9] https://github.com/qutebrowser/qutebrowser/issues/30

reviews of existing contributions, and deal with the merge conflicts after the research project finished as appropriate.

I decided to write this report in English so it is widely accessible by qutebrowser users and developers – I hope some of them will find the time to read it, and enjoy reading it as much as I enjoyed writing it. I initially was hesitant because I never wrote something of this scale in a foreign language before, but I quickly discovered that I was equally comfortable with writing English and writing German (my native language).

All in all, I am pleased with how things turned out, even if this is only the first step of many needed for proper extension support in qutebrowser. I hope I will soon be able to continue working towards this goal.

8.9. Acknowledgements

Working on an open-source project with an idea contributed by a student, as well as doing an SA as a single person, is rather untypical. I am very grateful that Prof. Stefan Keller was open to mentor this project as an advisor despite the unusual conditions. His support and insights in our regular meetings were invaluable.

Thanks to the qutebrowser community for making qutebrowser into far more than just a personal pet project of mine. Without this community, I doubt I would still be working on it after five years – when I started, I never imagined that it would get that far.

I am thankful to all qutebrowser contributors for their patience and understanding when I could not attend to their contributions in a timely manner while working on this project.

I would especially like to thank Fritz Reichwald for his contributions towards switching to the Sphinx documentation generator, even though I did not end up using his work in the scope of this project. We plan to meet up and merge the two approaches soon, so the entire qutebrowser documentation can be converted to Sphinx.

Thanks to AnneMarie O'Neill for her careful English reviews and constructive feedback on possible improvements. Her suggestions were always very welcome and accurate.

Last but not least, thanks to Méline Sieber for her constant encouragement throughout the semester, and for always helping me out of the maze when I got stuck somewhere. You were here when I needed it, *"für jede Meile bis zur letzten"*.

Appendices

A. Glossary and Abbreviations

API Application Programming Interface, i.e., in case of an extension API, the functions and classes (names, arguments, etc.) an extension can implement.

add-on A synonym for *extension*.

backend The software library doing the "heavy lifting" in qutebrowser, such as doing network requests or drawing website content.

CI Continuous integration, i.e., running automated tests or other checks with every change.

CSS Cascading Style Sheets, used to define styling/appearance of web pages.

decorator A *Python* decorator allows annotating/wrapping a function. See section 1.4.1.

DOM Document Object Model, the *API* to access a HTML document as a tree structure.

extension Code using an *API* in order to extend the functionality of an existing project.

GUI Graphical User Interface.

HSR The Hochschule für Technik (University of Applied Sciences) in Rapperswil, Switzerland.

HTML HyperText Markup Language, the language used to structure web pages.

IDE Integrated Development Environment – a software application providing tools (such as a source editor) for development.

IFS The Institute for Software at *HSR*.

JavaScript The programming language used for interactive web sites.

plugin In most contexts, the same as an *extension*. Note that qutebrowser initially used *plugin API* to refer to its extension API. However, this should be avoided, as "plugin" is too ambiguous in the context of web browsers: A plugin usually refers to software using the deprecated NPAPI (Netscape Plugin API) or PPAPI (Pepper Plugin API) technologies, such as Adobe Flash.

Python The programming language used to write qutebrowser.

PyQt A software library which allows to use *Qt* from *Python*.

Qt The *GUI* library used by qutebrowser.

QtWebEngine The *backend* used by default in qutebrowser, based on the Chromium project (like the Chrome web browser).

QtWebKit One of two possible *backends* qutebrowser can use. QtWebKit is the older backend, which is not used by default anymore, but is still supported.

SA Studienarbeit (student research project).

slot A slot is the method or function connected to a *signal* in *Qt*. See section 1.4.1.

signal A signal is a *Qt* feature which lets objects communicate with each other. See section 1.4.1.

W3C World Wide Web Consortium, the standards body for web-related technologies such as HTML.

B. API documentation

The API documentation on the following pages has been generated from Python docstrings in the API implementation. Sphinx 1.8.2 was used.

C. Task description / Aufgabenstellung

The original task description for this project (written in German) can be found on page 77ff.

CHAPTER ONE

API MODULES

1.1 cmdutils module

qutebrowser has the concept of functions which are exposed to the user as commands.

Creating a new command is straightforward:

```
from qutebrowser.api import cmdutils

@cmdutils.register(...)
def foo():
    ...
```

The commands arguments are automatically deduced by inspecting your function.

The types of the function arguments are inferred based on their default values, e.g., an argument *foo=True* will be converted to a flag *-f*/*–foo* in qutebrowser's commandline.

The type can be overridden using Python's function annotations:

```
@cmdutils.register(...)
def foo(bar: int, baz=True):
    ...
```

Possible values:

- A callable (`int`, `float`, etc.): Gets called to validate/convert the value.
- A python enum type: All members of the enum are possible values.
- A `typing.Union` of multiple types above: Any of these types are valid values, e.g., `typing.Union[str, int]`.

exception `qutebrowser.api.cmdutils.`**`CommandError`**

Raised when a command encounters an error while running.

If your command handler encounters an error and cannot continue, raise this exception with an appropriate error message:

```
raise cmdexc.CommandError("Message")
```

The message will then be shown in the qutebrowser status bar.

Note: You should only raise this exception while a command handler is run. Raising it at another point causes qutebrowser to crash due to an unhandled exception.

`qutebrowser.api.cmdutils.`**`check_overflow`**(*arg: int*, *ctype: str*) → None

Check if the given argument is in bounds for the given type.

Parameters

- **`arg`** – The argument to check.
- **`ctype`** – The C++/Qt type to check as a string ('int'/'int64').

`qutebrowser.api.cmdutils.`**`check_exclusive`**(*flags: Iterable[bool]*, *names: Iterable[str]*) → None

Check if only one flag is set with exclusive flags.

Raise a CommandError if not.

Parameters

- **`flags`** – The flag values to check.
- **`names`** – A list of names (corresponding to the flags argument).

`class` `qutebrowser.api.cmdutils.`**`register`**(**, instance: str = None, name: str = None, **kwargs*)

Decorator to register a new command handler.

`class` `qutebrowser.api.cmdutils.`**`argument`**(*argname: str*, ***kwargs*)

Decorator to customize an argument.

You can customize how an argument is handled using the `@cmdutils.argument` decorator *after* `@cmdutils.register`. This can, for example, be used to customize the flag an argument should get:

```
@cmdutils.register(...)
@cmdutils.argument('bar', flag='c')
def foo(bar):
    ...
```

For a `str` argument, you can restrict the allowed strings using `choices`:

```
@cmdutils.register(...)
@cmdutils.argument('bar', choices=['val1', 'val2'])
def foo(bar: str):
    ...
```

For `typing.Union` types, the given `choices` are only checked if other types (like `int`) don't match.

The following arguments are supported for `@cmdutils.argument`:

- `flag`: Customize the short flag (`-x`) the argument will get.
- `value`: Tell qutebrowser to fill the argument with special values:
  - `value=cmdutils.Value.count`: The `count` given by the user to the command.
  - `value=cmdutils.Value.win_id`: The window ID of the current window.
  - `value=cmdutils.Value.cur_tab`: The tab object which is currently focused.
- `completion`: A completion function to use when completing arguments for the given command.
- `choices`: The allowed string choices for the argument.

The name of an argument will always be the parameter name, with any trailing underscores stripped and underscores replaced by dashes.

`class` `qutebrowser.api.cmdutils.`**`KeyMode`**

Key input modes.

normal = 1
Normal mode (no mode was entered)

hint = 2
Hint mode (showing labels for links)

command = 3
Command mode (after pressing the colon key)

yesno = 4
Yes/No prompts

prompt = 5
Text prompts

insert = 6
Insert mode (passing through most keys)

passthrough = 7
Passthrough mode (passing through all keys)

caret = 8
Caret mode (moving cursor with keys)

`qutebrowser.api.cmdutils.`**`Value`**
alias of `qutebrowser.utils.usertypes.CommandValue`

1.2 apitypes module

A single tab.

class `qutebrowser.api.apitypes.`**`ClickTarget`**
How to open a clicked link.

normal = 0
Open the link in the current tab

tab = 1
Open the link in a new foreground tab

tab_bg = 2
Open the link in a new background tab

window = 3
Open the link in a new window

hover = 4
Only hover over the link

class `qutebrowser.api.apitypes.`**`InitContext`** (*data_dir*, *config_dir*, *args*)
Context an extension gets in its init hook.

class `qutebrowser.api.apitypes.`**`JsWorld`**
World/context to run JavaScript code in.

main = 1
Same world as the web page's JavaScript.

application = 2
Application world, used by qutebrowser internally.

user = 3
User world, currently not used.

jseval = 4
World used for the jseval-command.

`qutebrowser.api.apitypes.`**`Tab`**
alias of *`qutebrowser.browser.browsertab.AbstractTab`*

`qutebrowser.api.apitypes.`**`WebElemError`**
alias of *`qutebrowser.browser.webelem.Error`*

`qutebrowser.api.apitypes.`**`WebElement`**
alias of *`qutebrowser.browser.webelem.AbstractWebElement`*

exception `qutebrowser.api.apitypes.`**`WebTabError`**
Base class for various errors.

1.3 config module

Access to the qutebrowser configuration.

`qutebrowser.api.config.`**`val = None`**
Simplified access to config values using attribute acccess. For example, to access the `content.javascript.enabled` setting, you can do:

```
if config.val.content.javascript.enabled:
    ...
```

This also supports setting configuration values:

```
config.val.content.javascript.enabled = False
```

`qutebrowser.api.config.`**`get`**(*name: str*, *url: PyQt5.QtCore.QUrl = None*) → Any
Get a value from the config based on a string name.

1.4 downloads module

APIs related to downloading files.

class `qutebrowser.api.downloads.`**`TempDownload`**(*item: qutebrowser.browser.qtnetworkdownloads.DownloadItem*)
A download of some data into a file object.

`qutebrowser.api.downloads.`**`download_temp`**(*url: PyQt5.QtCore.QUrl*) → qutebrowser.api.downloads.TempDownload
Download the given URL into a file object.

The download is not saved to disk.

Returns a `TempDownload` object, which triggers a `finished` signal when the download has finished:

```
dl = downloads.download_temp(QUrl("https://www.example.com/"))
dl.finished.connect(functools.partial(on_download_finished, dl))
```

After the download has finished, its `successful` attribute can be checked to make sure it finished successfully. If so, its contents can be read by accessing the `fileobj` attribute:

```
def on_download_finished(download: downloads.TempDownload) -> None:
    if download.successful:
        print(download.fileobj.read())
        download.fileobj.close()
```

1.5 hook module

Hooks for extensions.

class `qutebrowser.api.hook.`**`init`**

Decorator to mark a function to run when initializing.

The decorated function gets called with a `qutebrowser.api.apitypes.InitContext` as argument.

Example:

```
@hook.init()
def init(_context):
    message.info("Extension initialized.")
```

class `qutebrowser.api.hook.`**`config_changed`**`(`*option_filter: str = None*`)`

Decorator to get notified about changed configs.

By default, the decorated function is called when any change in the config occurs:

```
@hook.config_changed()
def on_config_changed():
    ...
```

When an option name is passed, it's only called when the given option was changed:

```
@hook.config_changed('content.javascript.enabled')
def on_config_changed():
    ...
```

Alternatively, a part of an option name can be specified. In the following snippet, `on_config_changed` gets called when either `bindings.commands` or `bindings.key_mappings` have changed:

```
@hook.config_changed('bindings')
def on_config_changed():
    ...
```

1.6 interceptor module

APIs related to intercepting/blocking requests.

`qutebrowser.api.interceptor.`**`register`**`(`*interceptor: Callable[[qutebrowser.extensions.interceptors.Request], None]*`)` → None

Register a request interceptor.

Whenever a request happens, the interceptor gets called with a `Request` object.

Example:

```
def intercept(request: interceptor.Request) -> None:
    if request.request_url.host() == 'badhost.example.com':
        request.block()
```

class `qutebrowser.api.interceptor.`**`Request`**`(`*first_party_url, request_url, is_blocked=False*`)`

A request which can be intercepted/blocked.

`first_party_url = None`

The URL of the page being shown.

request_url = None
The URL of the file being requested.

block() → None
Block this request.

1.7 message module

Utilities to display messages above the status bar.

`qutebrowser.api.message.`**error**(*message: str*, ***, *stack: str = None*, *replace: bool = False*) → None
Display an error message.

Parameters

- **message** – The message to show.
- **stack** – The stack trace to show (if any).
- **replace** – Replace existing messages which are still being shown.

`qutebrowser.api.message.`**info**(*message: str*, ***, *replace: bool = False*) → None
Display an info message.

Parameters

- **message** – The message to show.
- **replace** – Replace existing messages which are still being shown.

`qutebrowser.api.message.`**warning**(*message: str*, ***, *replace: bool = False*) → None
Display a warning message.

Parameters

- **message** – The message to show.
- **replace** – Replace existing messages which are still being shown.

CHAPTER TWO

TAB API

class qutebrowser.browser.browsertab.**AbstractTab**
An adapter for QWebView/QWebEngineView representing a single tab.

window_close_requested
Signal emitted when a website requests to close this tab.

link_hovered
Signal emitted when a link is hovered (the hover text)

load_started
Signal emitted when a page started loading

load_progress
Signal emitted when a page is loading (progress percentage)

load_finished
Signal emitted when a page finished loading (success as bool)

icon_changed
Signal emitted when a page's favicon changed (icon as QIcon)

title_changed
Signal emitted when a page's title changed (new title as str)

new_tab_requested
Signal emitted when a new tab should be opened (url as QUrl)

url_changed
Signal emitted when a page's URL changed (url as QUrl)

contents_size_changed
Signal emitted when a tab's content size changed (new size as QSizeF)

fullscreen_requested
Signal emitted when a page requested full-screen (bool)

before_load_started
Signal emitted before load starts (URL as QUrl)

send_event (*evt: PyQt5.QtCore.QEvent*) → None
Send the given event to the underlying widget.

The event will be sent via QApplication.postEvent. Note that a posted event must not be re-used in any way!

fake_key_press (*key: PyQt5.QtCore.Qt.Key*, *modifier: PyQt5.QtCore.Qt.KeyboardModifier = 0*) → None
Send a fake key event to this tab.

dump_async(*callback: Callable[[str], None], *, plain: bool = False*) → None
Dump the current page's html asynchronously.

The given callback will be called with the result when dumping is complete.

run_js_async(*code: str, callback: Callable[[Any], None] = None, *, world: Union[qutebrowser.utils.usertypes.JsWorld, int] = None*) → None
Run javascript async.

The given callback will be called with the result when running JS is complete.

Parameters

- **code** – The javascript code to run.
- **callback** – The callback to call with the result, or None.
- **world** – A world ID (int or usertypes.JsWorld member) to run the JS in the main world or in another isolated world.

class `qutebrowser.browser.browsertab.`**AbstractAction**
Attribute `action` of AbstractTab for Qt WebActions.

exit_fullscreen() → None
Exit the fullscreen mode.

save_page() → None
Save the current page.

run_string(*name: str*) → None
Run a webaction based on its name.

show_source(*pygments: bool = False*) → None
Show the source of the current page in a new tab.

class `qutebrowser.browser.browsertab.`**AbstractPrinting**
Attribute `printing` of AbstractTab for printing the page.

check_pdf_support() → None
Check whether writing to PDFs is supported.

If it's not supported (by the current Qt version), a WebTabError is raised.

check_printer_support() → None
Check whether writing to a printer is supported.

If it's not supported (by the current Qt version), a WebTabError is raised.

check_preview_support() → None
Check whether showing a print preview is supported.

If it's not supported (by the current Qt version), a WebTabError is raised.

to_pdf(*filename: str*) → bool
Print the tab to a PDF with the given filename.

to_printer(*printer: PyQt5.QtPrintSupport.QPrinter, callback: Callable[[bool], None] = None*) → None
Print the tab.

Parameters

- **printer** – The QPrinter to print to.
- **callback** – Called with a boolean (True if printing succeeded, False otherwise)

show_dialog() → None
Print with a QPrintDialog.

class qutebrowser.browser.browsertab.**AbstractSearch**
Attribute `search` of AbstractTab for doing searches.

text
The last thing this view was searched for.

search_displayed
Whether we're currently displaying search results in this view.

_flags
The flags of the last search (needs to be set by subclasses).

_widget
The underlying WebView widget.

finished
Signal emitted when a search was finished (True if the text was found, False otherwise)

cleared
Signal emitted when an existing search was cleared.

search(*text: str, *, ignore_case: qutebrowser.utils.usertypes.IgnoreCase = <IgnoreCase.never: 2>, reverse: bool = False, result_cb: Callable[[bool], None] = None*) → None
Find the given text on the page.

Parameters

- **text** – The text to search for.
- **ignore_case** – Search case-insensitively.
- **reverse** – Reverse search direction.
- **result_cb** – Called with a bool indicating whether a match was found.

clear() → None
Clear the current search.

prev_result(**, result_cb: Callable[[bool], None] = None*) → None
Go to the previous result of the current search.

Parameters **result_cb** – Called with a bool indicating whether a match was found.

next_result(**, result_cb: Callable[[bool], None] = None*) → None
Go to the next result of the current search.

Parameters **result_cb** – Called with a bool indicating whether a match was found.

class qutebrowser.browser.browsertab.**AbstractZoom**
Attribute `zoom` of AbstractTab for controlling zoom.

apply_offset(*offset: int*) → None
Increase/Decrease the zoom level by the given offset.

Parameters **offset** – The offset in the zoom level list.

Returns The new zoom percentage.

set_factor(*factor: float, *, fuzzyval: bool = True*) → None
Zoom to a given zoom factor.

Parameters

- **factor** – The zoom factor as float.

- **`fuzzyval`** – Whether to set the NeighborLists fuzzyval.

class `qutebrowser.browser.browsertab.`**`AbstractCaret`**
Attribute `caret` of AbstractTab for caret browsing.

`selection_toggled`
Signal emitted when the selection was toggled. (argument - whether the selection is now active)

`follow_selected_done`
Emitted when a `follow_selection` action is done.

class `qutebrowser.browser.browsertab.`**`AbstractScroller`**
Attribute `scroller` of AbstractTab to manage scroll position.

`perc_changed`
Signal emitted when the scroll position changed (int, int)

`before_jump_requested`
Signal emitted before the user requested a jump. Used to set the special ‘ mark so the user can return.

class `qutebrowser.browser.browsertab.`**`AbstractHistory`**
The history attribute of a AbstractTab.

`back`(*count: int = 1*) → None
Go back in the tab’s history.

`forward`(*count: int = 1*) → None
Go forward in the tab’s history.

class `qutebrowser.browser.browsertab.`**`AbstractElements`**
Finding and handling of elements on the page.

`find_css`(*selector: str, callback: Callable[[Sequence[webelem.AbstractWebElement]], None], error_cb: Callable[[Exception], None], *, only_visible: bool = False*) → None
Find all HTML elements matching a given selector async.

If there’s an error, the callback is called with a webelem.Error instance.

Parameters

- **`callback`** – The callback to be called when the search finished.
- **`error_cb`** – The callback to be called when an error occurred.
- **`selector`** – The CSS selector to search for.
- **`only_visible`** – Only show elements which are visible on screen.

`find_id`(*elem_id: str, callback: Callable[[Optional[webelem.AbstractWebElement]], None]*) → None
Find the HTML element with the given ID async.

Parameters

- **`callback`** – The callback to be called when the search finished. Called with a WebEngineElement or None.
- **`elem_id`** – The ID to search for.

`find_focused`(*callback: Callable[[Optional[webelem.AbstractWebElement]], None]*) → None
Find the focused element on the page async.

Parameters **`callback`** – The callback to be called when the search finished. Called with a WebEngineElement or None.

find_at_pos(*pos: PyQt5.QtCore.QPoint, callback: Callable[[Optional[webelem.AbstractWebElement]], None]*) → None

Find the element at the given position async.

This is also called "hit test" elsewhere.

Parameters

- **pos** – The QPoint to get the element for.
- **callback** – The callback to be called when the search finished. Called with a WebEngineElement or None.

class qutebrowser.browser.browsertab.**AbstractAudio**

Handling of audio/muting for this tab.

set_muted(*muted: bool*, *override: bool = False*) → None

Set this tab as muted or not.

Parameters **override** – If set to True, muting/unmuting was done manually and overrides future automatic mute/unmute changes based on the URL.

is_recently_audible() → bool

Whether this tab has had audio playing recently.

CHAPTER

THREE

WEB ELEMENT API

class qutebrowser.browser.webelem.**AbstractWebElement**(*tab*)

A wrapper around QtWebKit/QtWebEngine web element.

tab

The tab associated with this element.

has_frame()

Check if this element has a valid frame attached.

geometry()

Get the geometry for this element.

classes()

Get a list of classes assigned to this element.

tag_name()

Get the tag name of this element.

The returned name will always be lower-case.

outer_xml()

Get the full HTML representation of this element.

value()

Get the value attribute for this element, or None.

set_value(*value*)

Set the element value.

dispatch_event(*event*, *bubbles=False*, *cancelable=False*, *composed=False*)

Dispatch an event to the element.

Parameters

- **bubbles** – Whether this event should bubble.
- **cancelable** – Whether this event can be cancelled.
- **composed** – Whether the event will trigger listeners outside of a shadow root.

insert_text(*text*)

Insert the given text into the element.

rect_on_view(***, *elem_geometry=None*, *no_js=False*)

Get the geometry of the element relative to the webview.

Parameters

- **elem_geometry** – The geometry of the element, or None.

- **no_js** – Fall back to the Python implementation.

is_writable()
Check whether an element is writable.

is_content_editable()
Check if an element has a contenteditable attribute.

Parameters **elem** – The QWebElement to check.

Returns True if the element has a contenteditable attribute, False otherwise.

is_editable(*strict=False*)
Check whether we should switch to insert mode for this element.

Parameters **strict** – Whether to do stricter checking so only fields where we can get the value match, for use with the :editor command.

Returns True if we should switch to insert mode, False otherwise.

is_text_input()
Check if this element is some kind of text box.

remove_blank_target()
Remove target from link.

resolve_url(*baseurl*)
Resolve the URL in the element's src/href attribute.

Parameters **baseurl** – The URL to base relative URLs on as QUrl.

Returns A QUrl with the absolute URL, or None.

is_link()
Return True if this AbstractWebElement is a link.

click(*click_target*, ***, *force_event=False*)
Simulate a click on the element.

Parameters

- **click_target** – A usertypes.ClickTarget member, what kind of click to simulate.
- **force_event** – Force generating a fake mouse event.

hover()
Simulate a mouse hover over the element.

class qutebrowser.browser.webelem.**Error**
Base class for WebElement errors.

class qutebrowser.browser.webelem.**OrphanedError**
Raised when a webelement's parent has vanished.

qutebrowser made extensible

- Studienarbeit im Herbstsemester 2018/19 Informatik
- Autor: Florian Bruhin
- Betreuer: Prof. Stefan Keller, Institut für Software, HSR
- Industriepartner: -

Aufgabenstellung

Projektvorstellung: Von der Projektseite: "qutebrowser is a keyboard-focused browser with a minimal GUI. It's based on Python and PyQt5 and free software, licensed under the GPL. Das Projekt existiert seit Dezember 2013 und hat inzwischen einige tausend User. Eine Extension-API, um qutebrowser mit eigenem Code zu erweitern, war bereits lange ein oft geäusserter Wunsch - da es aber schwierig ist, eine solche nachträglich zu ändern (ohne dass Extensions angepasst werden müssen), sollte sie gut geplant und mit Liebe zum Detail umgesetzt werden. Als wissenschaftlicher Aspekt soll ein eher theoretisch-orientiertes Kapitel zu Principles of API Design sowie Python Type Annotations erarbeitet werden.

Motivation: Viele Benutzer von qutebrowser sind power-user und haben dadurch spezifische Wünsche und Workflows. Es soll diesen Benutzern möglich gemacht werden, sich einfach ein Extension für die gewünschte Funktionalität zu schreiben, um den Kern von qutebrowser schlank zu halten. Ausserdem soll auch ein Teil des momentanen Kerns in Extensions ausgelagert werden (welche mit qutebrowser mitgeliefert werden).

Stand: Da qutebrowser nun rund 5 Jahre gewachsen ist und anfänglich ohne grosse Software-Engineering-Kenntnisse entwickelt wurde, hat sich eine gewisse "Technical Debt" entwickelt. Viel wurde im Laufe der Jahre schon aufgeräumt, aber einige Baustellen sind weiterhin offen. Diese sollen als Teil der Studientarbeit erst aufgeräumt werden, bevor eine Extension API entwickelt wird, um einen Einfluss auf die API zu vermeiden. Insbesondere sind dies folgende Punkte:

- Die Dokumentation von qutebrowser nutzt ein Tool (asciidoc) welches einige Probleme mit sich bringt und inzwischen nicht mehr weiter entwickelt wird. Um die Extension API auch adäquat dokumentieren zu können, soll auf Sphinx umgestiegen werden. Ein externer qutebrowser-Contributor (Fritz Reichwald) begann bereits vor der Studienarbeit mit ersten Arbeiten daran, und führt diese weiterhin fort. Er soll deshalb dabei unterstützt werden und es soll an seine Arbeit angeknüpft werden.
- Einige Stellen im qutebrowser-Code besitzen bereits Type Hints, aber mypy wird noch nicht systematisch (z.B. via Travis CI) eingesetzt. Dies soll geändert werden, und mindestens der Code, welcher über die Extension API exponiert wird, soll mit Type Hints versehen werden.

Vorgehen: Eine Extension API zu konzipieren birgt eine zentrale Frage: Wie mächtig soll die API sein und wie simpel - und dadurch stabil (im Sinne von "nicht ändernd") - wird sie? Dies wurde letztlich Firefox zum Verhängnis, denn das Extension-System von Firefox war mächtig, aber viel zu komplex und hat damit wichtige Änderungen an Firefox selbst erschwert. Als Resultat musste Firefox die XUL-API aufgeben und zwang damit alle bestehenden Add-ons zu einem grossen Rewrite.

In qutebrowser soll daher initial nur sehr wenig Funktionalität über die entsprechend sauber geplante Extension API exponiert werden. Es soll in agilen Zyklen gearbeitet werden, die die neue Extension-Funktionalität mit einem Refactoring von bestehendem Code verbinden:

- Via Use-Cases bzw. konkreten Extension-Ideen wird eine API für eine bestimmte Extension-Funktionalität ausgearbeitet.
- Diese API wird in qutebrowser implementiert.

- Bestehender Code im Core-Teil von qutebrowser, der ähnliche Funktionalität benutzt, wird umgeschrieben, um die Extension API zu benutzen. Als Beispiel: Nachdem eine API hinzugefügt wird, um Netzwerk-Requests zu blockieren, kann der in qutebrowser eingebaute Adblocker umgeschrieben werden, um diese API zu benutzen.

Lieferobjekte

1. qutebrowser refactored und ergänzt mit Extension API und internen Extensions
2. Zusätzliches Repository für Extensions.
3. Dokumentation (englisch) inkl. theoretischem Kapitel und dokumentiertem Extension API (API-Docs separat), Textabstract (englisch), Management Summary (englisch)
4. Software (englisch).
5. Die vom Studiengang geforderten bzw. empfohlenen Lieferobjekte: Poster (nur digital), Broschüren-Abstract, kein Kurzvideo.

Vorgaben/Rahmenbedingungen

- Grösstenteils gegeben durch das bestehende Projekt:
  - Python 3 (3.5+)
  - PyQt5 basierend auf Qt5 (5.7+)
  - pytest als Test Framework
  - Diverse Code Quality Tools (pylint/flake8/etc.)
  - Neu dazu kommen soll MyPy mit Type Annotations
- Allgemein:
  - Moderne SW-Entwicklung mit Unit Testing und kontinuierlicher Integration.
  - Nichtfunktionale Anforderungen: keine besonderen

Vorgehen und Arbeitsweise: Die Studierenden wählen nach Rücksprache ein Vorgehensmodell zur Softwareentwicklung. Es gibt wöchentliche Meetings mit vorbereiteten Unterlagen; wobei Ausnahmen vereinbart werden können.

Dokumentation

Zur Dokumentation (vgl. auch Lieferobjekte oben):

- Die Abgabe ist so zu gliedern, dass die obigen Inhalte klar erkenntlich und auffindbar sind (einheitliche Nummerierung).
- Die Zitate sind zu kennzeichnen, die Quelle ist anzugeben.
- Verwendete Dokumente und Literatur sind in einem Literaturverzeichnis aufzuführen (nicht ausschliesslich Wikipedia-Links auflisten).
- Dokumentation des Projektverlaufes, Planung etc.
- Weitere Dokumente (z.B. Kurzbeschreibung, Eigenständigkeitserklärung, Nutzungsrechte) gemäss Vorgaben des Studiengangs und Absprache mit dem Betreuer.

Form der Dokumentation zuhanden Betreuer (Studiengang siehe separate Instruktionen):

- Bericht gebunden (1 Exemplar)
- Alle Dokumente und Quellen der erstellten Software auf USB-Stick.

Bewertung

Es gelten die üblichen Regelungen zum Ablauf und zur Bewertung der Studienarbeit des Studiengangs Informatik mit besonderem Gewicht auf moderne Softwareentwicklung wie folgt:

- Projektorganisation (Gewichtung ca. 1/5)
- Bericht, Gliederung, Sprache (Gewichtung ca. 1/5)
- Inhalt inkl. Code (Gewichtung ca. 2/5)
- Gesamteindruck inkl. Kommunikation mit Industriepartner oder weiteren Beteiligten (Gewichtung ca. 1/5).

Ein wichtiger Bestandteil der Arbeit ist, dass eine lauffähige, getestete Software abgeliefert wird (inkl. Softwaredokumentation).

Weitere Beteiligte

Freiwillige aus der qutebrowser-Community.

D. Urheber- und Nutzungsrechte

Die Urheber- und Nutzungsrechte an der Software dieser Studierendenarbeit (nachfolgend Arbeit genannt) sind im entsprechenden Software-Repository geregelt. Die Urheber- und Nutzungsrechte aller anderen Artefakte der Arbeit sind wie folgt geregelt: Die Schutzrechte und das Know-how an den in diesem Rahmen geschaffenen Gütern (ausser Software, vgl. oben), stehen sowohl dem Rechtsträger der HSR Hochschule für Technik Rapperswil, dem für die Arbeit verantwortlichen Professoren, den allfällig beteiligten Projektpartnern, sowie den Verfassern der Arbeit resp. Entwickler der in diesem Rahmen geschaffenen Güter zu. Die genannten Parteien übertragen sich gegenseitig nicht exklusiv, jedoch unentgeltlich, weltweit, sachlich und zeitlich unbeschränkt die jeweiligen Schutzrechte und das Know-how an der Arbeit und an der in diesem Rahmen geschaffenen Güter, einschliesslich dem Recht zur Weiterübertragung, ab. Entsprechend steht es jeder Partei zu, sämtliche Schutzrechte an der Arbeit resp. an der in diesem Rahmen geschaffenen Güter beliebig weltweit, zeitlich und sachlich unbeschränkt zu verwerten. Darunter fällt namentlich aber nicht abschliessend das Recht zur Lizenzierung in jeder Art, Umfang und Form, das Recht zur Bearbeitung und damit zur Nutzung als Grundlage eines neuen schutzfähigen Guts. Die Parteien erklären sich gegenseitig den Verzicht auf Namensnennung bei der Verwertung der Schutzrechte und des Know-how durch eine oder mehrere Parteien gemeinsam und stimmen namentlich zu, dass jede Partei allein unter ihrem eigenem Namen die Schutzrechte resp. das Know-how verwertet. Die vorliegende gegenseitige unentgeltliche Übertragung der Schutzrechte resp. des Know-how bezieht sich auch auf Verwertungsarten, welche heute noch nicht bekannt sind.

E. Test log

```
========================= test session starts ==========================
platform linux - Python 3.7.1, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
cachedir: .tox/py37/.pytest_cache
PyQt5 5.11.3 - Qt runtime 5.12.0 - Qt compiled 5.12.0
benchmark: 3.1.1 (defaults: [...])
hypothesis profile 'default' -> [...]
rootdir: /home/florian/proj/qutebrowser/git, inifile: pytest.ini
plugins: xvfb-1.1.0, travis-fold-1.3.0, rerunfailures-5.0, repeat-0.7.0,
  qt-3.2.1, mock-1.10.0, instafail-0.4.0, faulthandler-1.5.0, cov-2.6.0,
  benchmark-3.1.1, bdd-3.0.0, hypothesis-3.82.5
collected 195 items

tests/unit/api/test_cmdutils.py ..........................................
                                ...................
tests/unit/components/test_adblock.py ..............................
tests/unit/components/test_misccommands.py .....
tests/unit/extensions/test_loader.py ............
tests/unit/browser/webengine/test_webenginetab.py .....
tests/end2end/features/test_caret_bdd.py ...........
tests/end2end/features/test_scroll_bdd.py ..................s.............
                                          ...................
tests/end2end/features/test_zoom_bdd.py ...............x....


-------- benchmark: 1 tests --------
Name (time in us)            Min      Max  Median
-------------------------
test_adblock_benchmark     3.3040  40.3980  3.4670
-------------------------

Legend:
  Outliers: 1 Standard Deviation from Mean; 1.5 IQR (InterQuartile Range)
            from 1st Quartile and 3rd Quartile.
  OPS: Operations Per Second, computed as 1 / Mean
=========== 193 passed, 1 skipped, 1 xfailed in 44.72 seconds ===========
```

Tests marked with *s*/*skipped* were skipped because of platform/environment differences. Tests marked with *x*/*xfailed* were expected to fail due to known bugs.

Note only the subset of tests relevant to this project was ran.

F. Literature and Sources

Batchelder, Ned (2012). *Eval really is dangerous*. URL: https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html (visited on 10/29/2018).

Blanchette, Jasmin (2008). *The Little Manual of API Design*. URL: https://people.mpi-inf.mpg.de/~jblanche/api-design.pdf (visited on 11/20/2018).

Buschmann, F. et al. (1996). *Pattern-Oriented Software Architecture, A System of Patterns*. Pattern-Oriented Software Architecture. Wiley. ISBN: 9781118725269.

Conca, Mike (2018). *Firefox, Chrome and the Future of Trustworthy Extensions*. URL: https://blog.mozilla.org/addons/2018/10/26/firefox-chrome-and-the-future-of-trustworthy-extensions/ (visited on 10/26/2018).

Edge, Jake (2013). *The failure of pysandbox*. URL: https://lwn.net/Articles/574215/ (visited on 10/29/2018).

GitHub, Inc. (2017). *Open Source Survey*. URL: https://opensourcesurvey.org/2017/ (visited on 12/20/2018).

Hochschulrat (2015). *Richtlinien des Hochschulrates für die koordinierte Erneuerung der Lehre an den universitären Hochschulen der Schweiz im Rahmen des Bologna-Prozesses*. URL: https://www.admin.ch/opc/de/classified-compilation/20150869/index.html (visited on 11/12/2018).

Johnson, R. et al. (2005). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley. ISBN: 9780201633610.

Needham, Kev (2015). *The Future of Developing Firefox Add-ons*. URL: https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/ (visited on 10/29/2018).

Pietraszak, Mike (2017). *Browser Extensions - Draft Community Group Report 23 July 2017*. Tech. rep. W3C and Microsoft Corporation. URL: https://browserext.github.io/browserext/ (visited on 10/26/2018).

Veditz, Daniel (2015). *The Case for Extension Signing*. URL: https://blog.mozilla.org/addons/2015/04/15/the-case-for-extension-signing/ (visited on 10/29/2018).

Zimmermann, Olaf (2012). *Making Architectural Knowledge Sustainable – Industrial Practice Report and Outlook*. Invited IEEE Software Speaker, SEI SATURN. ABB Corporate Research. URL: https://resources.sei.cmu.edu/library/asset-view.cfm?assetID=22132 (visited on 12/19/2018).