Coding Choices

Octopeer for Bitbucket

In this document some of the choices we've made during the course of the project are clarified.

Adding dependencies in our project instead of linking

We add the dependencies for jQuery and Materialize (the minimized versions) directly in our extension when building it instead of linking to the original source. We understand that any fixes won't be taken into account for our extension but we wanted to reduce the security risks as we would have to give permissions to access these specific websites.

Code Duplication

In the trackers there is a little bit of duplicate code we are aware off. The language we use in combination with the tools makes it require a restructuring of our tooling system to fix this issue. When developers have time this is an issue that we recommend to be fixed to improve maintainability.of the product.

DomTracker design choices

When creating the DomTracker, we chose for the change event which fires in specific cases but afterwards we found out that it didn't fire when a few very important events were fired. Solving the problem of registering the few events we didn't already catch, we started using the MutationObserver. The MutationObserver did indeed fire at the events as we wanted, but it also fired when we made a change to the dom for adding the coords to every element. These DOM changes were needed as we found no solution in creating a hard-copy of the DOM without destroying the performance of the extension. So this made the MutationObserver useless. As a second solution we looked at what specific events were not been registered yet, these events were all triggered by a click on the same tag types. As solution we put individual addListeners on every item with these specific tags which solved our problem.

ES5 vs. ES6

We are using an outdated version of ECMAScript as some features are not supported by the TypeScript compiler at this moment and not all features are supported by the Chrome browser. To avoid the usage of these features we are using the most stable version.

Foldername ‘JS’ instead of ‘TS’ for TypeScript files

It is a convention to call the folder which consists of TypeScript files ‘JS’, the abbreviation of JavaScript, as after transpiling, there will only be JavaScript files left. The reason we choose for this convention is that renaming and rewriting paths during build overly complicates things while not adding much benefit.

Global Code

The trackers we use are injected into the webpage, without a single line of global code, only classes and functions will get injected, but won't get used. We need a few lines of global code to start using the logic contained inside the classes. The global code works like an ignition in a car, "starting up" the code inside the classes and functions.

Global Mocks

In order to test properly we need to mock several functions, including some chrome functionality. We discovered that global mocks in separate files affect each other, as this lead to failing tests. Since there is also some code duplication we decided to group all global mocks in a single file.

Keylogger

When logging keyboard events there are multiple events to choose from. *keydown*, *keypress*, or *keyup*. We chose *keyup* event in our keylogger. The benefit of choosing the *keyup* event over the *keypress* is that the event is also triggered when a special key (e.g. Page Up, Control) is pressed. There is also a small drawback, the *keyup* event handles all characters as capitals, regardless of Caps Lock or Shift usage. We thought this was less of a handicap than leaving special keys out though.

Interfaces for database schemes instead of classes

The idea behind this is to have the static typechecker to check against the server API documents, so that we have static checks performed on the correctness of the messages.
We use them to validate the format of the object - and therefore the stringified form, JSON - to make sure we dot not make stupid mistakes. The interfaces are thus not interfaces as in the usual language interface style.

This solution does make the data objects less open to extension, but since these are data storage objects it isn’t necessary as there is a very little chance this will change.

Mouse position / throttling

When adding a *mouse mouse* event listener it will fire a lot. If data is send to the database every time an event occurs, the database might be flooded. Because of this we decided to throttle the amount of times data is send to a maximum of one time per second. At this points any superfluous data is discarded. We will look into implementing a strategy design pattern for the throttling, so that we may swap it out for a different solution (such as bulk sending messages) easily in the future.

Setting.ts / SettingsExplanations.ts self initialisation

We aim to initialize most trackers and other functionality via the main or background files. This is a little different for the settings and settings explanation files. They are not used inside the extension, but are loaded into an html page as separate scripts. The code had to be encapsulated in classes to make it testable (and a bit more neat), but this way it needs to be called for the javascript to start working. That had to happen in the files themselves, because the are loaded under different circumstances than the rest of the extension.

Testing Coverage

While we aim to keep our test coverage above 85% at all times, getting 100% test coverage is very hard. Some files are very hard or impractical to test. Below we've explained the reasons why test coverage is particularly low in some files.

Background.ts
This file only contains code to let the settings page become visible in a new tab. We think it is better to run a manual test on this to verify correct behaviour, also as there can only be two kinds of behaviour: '1. It works or 2. It doesn't work'.

As this file has increased in size, we're looking to test it in the coming weeks.

Main.ts
This file has not been tested and is used to run the extension's trackers. As this file makes heavy use of the Chrome browser local storage which we globally mock in our tests, we could only be testing a mock which makes the tests useless. Instead we prefer to run a manual test to verify the correct behaviour.

OctopeerConstants.ts
As this file only contains constants for enhanced maintainability, we came to the conclusion that this file doesn't have to be tested.

RARequestSender.ts
This file has only be tested for 88%. The part that hasn't been tested makes use of the chrome messaging API. As we globally mock this code to test the behaviour of other classes, we are unable to test the piece of code in this file. Instead, we verify the behaviour manually as it is concerned with data getting stored in the database, which can easily be verified by checking the database.

Settings.ts
The loading of flipbox values is a trivial task, and therefore we did not deem it necessary to write a separate test for this.

SettingsExplanations.ts
This file handles the live demos for the settings page and has a line coverage of 80%. Although we were able to test most code, testing the anonymous event listeners is quite a bit harder. We didn't think spending the extra effort for just testing the event listeners was worth the extra effort. If they don't work, the UI doesn't work, so problems would be noticed quite quickly. A manual test will be good enough for this we thought.

TestSemanticTracker.ts
The semantic tracker test has not full coverage which should actually never happen. This is due to the way we are writing some tests. It requires a new class to be implemented which extends the semantic tracker. The extending enforces the creation of a method which we don't use in the tests which is why we don't achieve full test coverage in a test file.

Union Operator in Message Interface

In the Message interface, an union operator is used. It has the same appearance as the OR operator in other programming languages but in typescript is has a side-effect. We use the Message interface for type-checking purposes. The union operator does a correct type-check on the database endpoint locations we defined but when applying this operator on other interfaces the side-effect appears. This side-effect means that not only these specific interfaces are allowed but also interfaces that are made up by components of other interfaces, which is the edge case.

Although we know about this side-effect, we stick to using this interface for type-checking purposes as it correctly type-checks the endpoints and in most cases it correctly type-checks the interfaces well. Only not in the edge cases.