Code modularity

Note: this page needs to be updated to reflect changes made in Haketilo 1.0-beta1

Complex functionality has to be split into multiple JavaScript files. However, managing these by hand can get very cumbersome as project grows. There exist various ways of bundling JavaScript. Some tools commonly used for this purpose are Browserify, jspm, Webpack, RequireJS and Babel. Yet another option is to use JavaScript modules - a native feature of the language.

Native modules considerations

JavaScript modules are rather convenient to use. Unfortunately, support varies between browsers and in some cases there is simply no facility to load a module. An example of such case are content scripts. They are loaded as listed from manifest.json file and need to be traditional script files. Chromium already offers a way to load a module dynamically from within a traditional script, but all modules loaded this way (including those pulled by the initial one) need to be listed as web-accessible resources in the manifest.

Bundler considerations

If we decided to use a bundler, it would create an additional dependency for the project. Haketilo clearly has no need for many of the features bundlers provide and as long as the functionality needed from them can be provided by a simple script, this is preferred.

It is also worth mentioning some of the bundlers are part of the Node.js/NPM ecosystem which, although consisting mostly of libre software, is problematic from software freedom point of view due to requiring people to rely on a third-party package manager and a non FSDG-compliant software repository. If any bundler is to be used, it is preferred to choose one that doesn't rely on Node.js or at least is packaged for an FSDG-compliant GNU/Linux distribution.

Current approach

At some point early on, this extension utilized native JavaScript modules. Due to problems with these, sources were slightly modified, so that the import/export semantics of modules were mostly retained, but import and export statements were replaced with assignments and accesses to keys of the window object. Each script file's contents are wrapped into a function call so as not to clobber the global namespace. Properly ordered lists of scripts to load in each context were compiled by hand and placed in manifest.json and relevant HTML files.

The build.sh script is now used to automatically write out such assignments and accesses to keys of the window.killtheweb object, wrap scripts, and compile those lists. Imports are performed using comments of the form

```
/*
 * IMPORTS_START
 * IMPORT first_import
 ...
 * IMPORT last_import
 * IMPORTS_END
 */
```

placed before the beginning of the code, while exports use a similar structure after the code with EXPORT replacing IMPORT throughout.