BibItNow! Site Adjusters – How to contribute?

Langenscheiss

April 24, 2018

Contents

Welcome

Hej.
If you have decided to contribute by writing your own site adjusters, thanks a lot! This is really going to help me! So let me return the favor and help you getting started with this little step-by-step guide.

1 Step 1 – Get full source code

The code in this github repository only exposes the parts of the plugin for which I currently (you may always inspire me to change this policy) accept external contributions. However, for testing/debugging purposes, or for figuring out how the code works if you wish, you obviously need the full source code. You can download the latest developer versions for all currently supported browsers from my website.
Since site adjusters work browser-independently, you may pick whatever browser-variant you prefer as your own developer version. In the following, we will denote the local plugin root directory, i.e., where *manifest.json* is located, as "$PLUGINDIR". The corresponding root directory in the github repository will be called "$GITDIR".

2 Step 2 – Enabled desired site

The next step is to add the desired website to the plugin. Open the Adjuster List located in

$PLUGINDIR/nameResources/urlSpecificAdjusterList.json

, written in the JSON format. Add an adjuster entry to this file. The structure is pretty self-explanatory. Specify the URL scheme, and the filenames for the adjuster scripts. For example, if you wish to add the website "https://www.johndoe.com", and you wish to link it to script adjuster files "johndoe.js" (both for preformatter and prefselector), you add the object

```
{ "scheme": "johndoe" , "top": "com" ,
  "prefselector": "johndoe" , "preformatter": "johndoe"}
```

to the JSON array. Note that the names of script files are stated **without the .js extension**. Details about how to specify more complex matching schemes, and how these schemes are then matched with the found URL are described in App. C.

Once you have added an entry to the adjuster list, linking an URL scheme to preformatter/prefselector script files, you need to make sure these files exist, as the plugin will crash otherwise if the URL scheme is positively matched. If you have specified "johndoe" as preformatter, add a copy of "0_TEMPLATE.js" from the preformatter directory in the github repository

$GITDIR/background/preformatters/

to

$PLUGINDIR/background/preformatters/

and rename it to "johndoe.js". Follow the same procedure for the "prefselector" specified in the prefselector file, with "0_TEMPLATE.js" now of course taken from

$GITDIR/extractors/prefselectors/

and copied to

```
$PLUGINDIR/extractors/prefselectors/
```

.

These 3 steps should be enough to let the plugin know that a prefselector and preformatter should be loaded for the given website. If you haven't adjusted the template files, and everything is correct, the global web console should show the message "This seems to work!" if you activate the plugin popup while surfing on the given website.

3 Step 3 – Find optimal prefselectors

Now, the potentially more difficult part starts. You need to study the source of abstract/article article pages in order to figure out CSS selectors which select the desired citation info to be read by the extractor. The first thing to look for are meta tags. Decent publishing houses put the most relevant citation info into such meta tags. Hence, **BibItNow!** has a fixed kernel of search queries for most bibliography fields – henceforth shortened as "bibfields" – and if you are lucky, those queries are already enough to complete a citation (details about the bibfields and their corresponding default search queries are presented in App. A). However, often you are not that lucky, and you need to be more inventive in providing custom CSS selectors for search queries. Technically, these search queries are carried out PRIOR to those defined in the fixed kernel, thereby suggesting the name "preferred selector" or "prefselector" as a short form.

Once you think you have found the info in the website HTML code[1] and the corresponding CSS selector for, e.g., the author(s) of an article, refer to App. D and to the example file

```
$GITDIR/extractors/prefselectors/0_EXAMPLE.js
```

in order to understand how to precisely link a CSS selector to a certain bibfield, which in this case would be *citation_authors*.

The basic procedure is to provide, as first argument, a CSS selector selecting the desired HTML element, and as second argument, an attribute of the selected element that should be read out and saved for the corresponding bibfield. You can add as many custom queries as you want, and each query can be further specified with several optional arguments next to the CSS selector and the attribute. For example, if you have found the author info in the two non-standard meta tags

```
<meta name="bla_author" content="John Doe"></meta>
<meta name="bla_author" content="Jane Doe"></meta>
```

which are not recognized by the fixed kernel, you will have to add the property

[1] Note that **BibItNow!** queries code **after** the page has finished loading, possibly including effects of dynamically loaded scripts on the website's DOM. The static source code might hence not always be representative of the information that is available to the plugin. In other words, be sure to inspect the final DOM in case a CSS selector does not find the desired information.

```
citation_authors : [['meta[name="bla_author"]','content']]
```

that is, with CSS selector “meta[name="bla_author"]” and attribute “content” (since the “content” attribute of the meta tags contains the data) to the JSON object *prefselectorMsg* in the prefselector script file.

As explained in App. A.1 for the author bibfield, the extractor will select EVERY ELEMENT that is found using the specified prefselector string, and concatenates everything into a semicolon-separated list, as the main parser expects this format. So, if the above author information was instead given in a single tag such as

```
<meta name="bla_author"
        content="John Doe and Jane Doe"></meta>
```

you will first have to replace the ”and“ by a semicolon. This is one main reason why **BibItNow!** has preformatters, see Sec. 5. The file “0_EXAMPLE.js“ shows more examples of adding prefselectors. As a general rule, try to use CSS selectors which read the info in a robust way. Remember that websites change from time to time.

4 Step 4 – Parse a link for dynamic citation export if possible

While not necessarily something for your first shot at writing a site adjuster, remember that a core functionality of **BibItNow!** is to communicate with the citation export/”Download citation“-button offered on abstract pages of most publishers/databases. In the absolute majority of all cases, these buttons are technically form submission buttons or simple file links, i.e., something that can be called with an XHR to a specifically formatted URL. Since the main parser is expecting the downloaded citation to be in the RIS citation format, it is highly recommended – yet not necessary – to parse a URL which links to a resource in this RIS format.

To determine the URL per citation, **BibItNow!** provides 2 stages:

1. In the *prefselectorMsg* object, the *citation_download* property allows to query a download link from the abstract page.

2. If such a download link is found, this link together with all extracted static data is passed to the second stage – the *formatCitationLink* function defined in the prefselector script file. Possibly using all static citation data including a citation URL, this function allows to specify the XHR method (GET or POST) and requires you to return a formatted and finalized request URL. **Note** that if an invalid URL is returned, or if **no preformatting script has been found for the website**, the plugin will skip the dynamic citation download request altogether. If, however, a request is sent, and if it finishes with status code 200 (OK, successful), the response data will be saved as text into the *citation_download* property of the *metaData* object accessible in preformatting.

There are a number of important rules to obey in using the static citation data, and in parsing the request URL.

1. **Rule 1: Never parse any data to anything but text.** It is sometimes tempting to use *eval* on data, or to assign it to the *innerHTML* property of a DOM node. However, for security reasons, this will not be tolerated in any code of **BibItNow!**, including site adjusters. In particular, Mozilla warns against this practice when submitting web extensions. It is allowed to ”try-catch“ a *JSON.parse* on text data, as the content of the returned JSON object is interpreted as data only. However, the author generally advises against parsing with anything but string methods if it does not require too much extra code. Reading, e.g., a single property from a JSON string can often be done with a simple regular expression; it does not require you to parse the whole string into a JSON object. Note also that performance is currently not a bottle neck for **BibItNow!**, so you can always afford to manually parse raw strings with regexp magic.

2. **Rule 2: Avoid cross-site and mixed-content requests.** Modern browsers prohibit cross-site and mixed-content XHR. In other words, for the XHR to be successful, you need to stay on the same domain as in the active tab, from which you have extracted the static citation data, and you may not switch between ”http“ and ”https“. There is only one exception allowed by **BibItNow!** : cross-site requests to ”citation-needed.springer“. This exception currently exists because the publisher ”Nature“ is in the process of merging with the publisher ”Springer“.

Refer to the example file ”0_EXAMPLE.js“ for a demonstration of how to successfully parse a citation download link.

5 Step 5 – The most important step: preformatting

As already mentioned in Sec. 3, the raw data extracted from the website source is, in many cases, not immediately ready to be understood by the main parser, and sometimes not even available at all (which means you need to hardcode it). As the name suggests, the purpose of the preformatting stage is to preformat the data and to correct all these flaws before the main parser takes over. For more details and ”hands-on“ instructions, refer to the file ”0_EXAMPLE.js“ in

$GITDIR/background/preformatters/

In the above mentioned example of two author names provided in one meta tag, but not in a semicolon-separated list, the *preformatData* function in the preformatting script file would have to contain a line similar to

```
metaData["citation_authors"] = metaData["citation_authors"]
                                .replace(/[\s]+and[\s]+/gi," ; ");
```

in order to correct for this mistake. The example file illustrates more complex modifications, and App. A explains which bibfield expects precisely which data structure in the main parser. **Note carefully** that the same **rules and restrictions to parsing data as stated in Sec. 4** also apply to the entire preformatting stage.

The preformatting stage is divided into 2 functions that are called in the following sequence:

1. **The function *preformatRawData*** is only called if the dynamic citation download request yielded a positive response with valid, non-empty response data. The *citation_download* property of the *metaData* JSON-object passed to this function then contains the raw response text. In the subsequent parser stage, this text is assumed to represent citation data in the RIS-format! Hence, if this is not already the case at this stage, you need to reformat the data by modifying it in accordance with the restrictions stated in Sec. 4 (see the site adjuster for PubMed as an example of how to deal with this situation!).

2. **The function *preformatData*** is called in any case. If any data from the dynamic citation download could – after calling *preformatRawData* – be successfully parsed, it will be accessible as a JSON object linked to in the *citation_download* property of the *metaData* object. The bibfields in this object are associated with exactly the same properties as in the *metaData* object itself. For example,

   ```
   metaData["citation_title"]
   ```

   contains the title of the citation as obtained from the static data, while

   ```
   metaData["citation_download"]["citation_title"]
   ```

   contains the title obtained from the dynamic download request in case the latter was successful. **Importantly**, after the *preformatData* function has returned (it returns void), any non-empty string in the JSON object linked to in the *citation_download* property will replace the corresponding static data. In other words, if you, for example, want the plugin to prefer the statically obtained citation title, you will have to add a code similar to

   ```
   if (metaData["citation_title"] != "") {
           let download = metaData["citation_download"];
           if (download != null
                   && typeof(download) == 'object') {
                       download["citation_title"] = "";
           }
   }
   ```

   to the *preformatData* function in order to erase the dynamically obtained data. Note that since *metaData["citation_download"]* is only an (empty) string if no data was obtained through a dynamic download, you always need to properly check for that to avoid crashes!

6 Step 6 – Debugging and Submission

Once you have written everything, continue testing your site adjuster with various sources on the website of interest. Typically, one overlooks edge cases that need adjustments either in the definition of the preferred selectors, or in the preformatting stage. The more robust your adjuster is, the better.

Finally, once everything is ready, use the github repository to propose a new addition. You do not need to upload a new version of the URL adjuster list. Simply state the URL scheme in the header of your adjuster script files, and the adjuster will be added for the next adjuster upgrade release (which may appear in a different frequency compared to feature updates), given that all criteria for a correct submission are fulfilled.

THANKS!

A Bibfields

In this appendix, we list all bibliography fields (bibfields) accessible via preferred selectors and/or in the preformatting stage. More precisely, subsection A.2 states, for each bibfield, the purpose for the final formatted citation, the format and data type expected or enforced during and at the end of the preformatting stage, and what in the following subsection A.1 is defined as the provided *extraction class*. The fixed default set of CSS selectors and attributes used to search for bibfield data in the DOM (see Sec. 3) can be found in the example prefselector file in the github repository.

A.1 Extraction class

Depending on the bibfield and its specific needs and purpose for the final citation, **BibItNow!** adjusts the precise way in which it uses CSS selectors to extract data from the HTML source. This concerns mainly how much data is extracted (how many search queries are performed), and in which format this data is passed onto the parser/preformatting stage. For example, to obtain *all* author name information, it makes sense to not finish a search query with the first non-empty hit, but to instead search for all elements selected by a particular CSS selector.

Altogether, **BibItNow!** differentiates between bibfields in 4 **extraction classes**.

- **Class 1:** For each bibfield belonging to class 1, the plugin stops the search as initiated by the (default or custom, see Sec. 3) CSS selectors after the first non-empty string is found. Any bibfield of class 1 supports custom, site-specific CSS selectors defined as described in section 3 and App. D.

- **Class 2a:** For bibfields in class 2a, the plugin searches until the first CSS selector (default or custom) gives a non-empty result, then searches and reads all elements selected by this CSS selector and the associated attribute. A bibfield-dependent maximum number of result strings are then concatenated to one string containing a semicolon-separated list and sent to the preformatting stage/main parser. Just as for class 1, all bibfields which are part of this class support custom CSS selectors, attributes, etc. , including the possibility to reset the maximum number of result strings passed on.

- **Class 2b:** The only difference to class 2a is that the search does not stop with the first CSS selector/attribute yielding a non-empty result. Instead, all default and and all custom selectors are queried, and all results are concatenated to a string containing a semicolon-separated list.
- **Class 3:** These are additional bibfields which are accessible in the preformatting stage, but for which the data extraction cannot be guided by preferred selectors.

Regardless of its class, the extracted string of any bibfield is passed in a sanitized form to the parser/preformatting stage, where the bibfield data is available through the *metaData* object, see Sec. 5. As detailed in App. B and App. D, custom prefselectors give limited control over this sanitizing process.

A.2 List of bibfields

Let us now describe each bibfield. The following list is sorted according to the 4 extraction classes defined in Sec. A.1. For each bibfield, the **format** and **content** both in the raw preformatting (RPF) and preformatting (PF) stage (see Sec. 5) are stated. The format is always the **expected format before entering the next parsing stage**, which for raw preformatting is the preformatting stage, and for the preformatting stage the final parsing stage. If the information contained in the bibfield, i.e., the data accessible via

```
        metaData["#BIBFIELD"]
```

, with #BIBFIELD being one of the following bibfields, does not follow the format restrictions expected for the next stage, the system will try to reformat it to the best of its abilities – by, e.g., removing forbidden characters – but parsing may fail in those cases. One main purpose of preformatting is thus to ensure that citation data is available to the main parser in the standard format.

A.2.1 Class 1

- citation_publisher
  - **Format:** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Publisher of cited work, e. g., ”Wiley“ or ”Elsevier“
- citation_journal_title
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Title of the journal in which the cited work appears, i.e., ”Science“ or ”Nature“.

- citation_journal_abbrev
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Abbreviation (as obtained from the source!) of the title of the journal in which the cited work appears, i.e., ”Nat. Comm.“ for ”Nature Communications“. If available, the plugin prefers the abbreviation from its own abbreviation database.
- citation_volume
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Volume (number) of cited source.
- citation_issue
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Issue (number) of cited source.
- citation_date
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** String that can unambiguously be parsed to a date. Preferrably, but not necessarily, in YYYY-MM-DD format.
- citation_archive_id
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Citation ID in open access archive (such as arXiv).
- citation_abstract
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Abstract of citation.
- citation_issn
  - **Format (RPF, PF)** Single string containing the characters ”0-9“,”X“, and ”-“
  - **Content (RPF, PF):** ISSN of series in which cited source is published.
- citation_isbn

  - **Format (RPF, PF)** Single string containing the characters ”0-9“,”X“, and ”-“
  - **Content (RPF, PF):** ISBN of cited book/thesis.

- citation_firstpage
  - **Format (RPF, PF)** Single string containing ASCII characters
  - **Content (RPF, PF):** Indicates first page within the work in which citation is published.

- citation_lastpage
  - **Format (RPF, PF)** Single string containing ASCII characters
  - **Content (RPF, PF):** Indicates last page within the work in which citation is published.

- citation_url
  - **Format (RPF, PF)** Single string containing URI compatible characters
  - **Content (RPF, PF):** URL of citation source website. By default, this is the URL in the address bar.

- citation_doi
  - **Format (RPF, PF)** Single string containing URI compatible characters
  - **Content (RPF, PF):** DOI of citation.

- citation_title
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Contains the title of cited work, i.e., title of the book, scientific article, thesis, etc. .

- citation_type
  - **Format (RPF, PF)** Single string containing any unicode characters which survive sanitizing
  - **Content (RPF, PF):** Contains any keyword that could indicate the type of cited work. After the preformatting stage the system will look for keywords in this bibfield in order to determine whether the source is an article, a book, a thesis, or a generic source (websites etc.).

TODO CONTINUE

B Static extraction sanitizer

TODO

C URL Matching

In this appendix, we first describe the format of URL specific adjuster list, located in

```
$PLUGINDIR/nameResources/urlSpecificAdjusterList.json
```

, and then explain how the URL schemes specified in this list are used for URL matching.

C.1 URL List Format

The URL list is essentially a single array of JSON objects, each corresponding to a URL scheme that is potentially matched to the URL of the website on which the plugin is launched. Each object contains up to 5 valid properties – ”scheme“, ”top“, ”path“, ”prefselector“ and ”preformatter“ – to which either a string or, in some cases, another array of JSON objects can be assigned. An example of such a JSON object that **BibItNow!** uses in order to recognize and load site adjusters for the ”Science“ Journals is

```
{ "scheme": "(?:|[0-9a-z\\-]+[\\.]+)sciencemag" ,
  "top": "org" ,
  "prefselector": "science" , "preformatter": "science" },
```

Let us describe the properties in detail:

1. **Property:** ”scheme“ – **Type:** *String* – **Required:** yes – **Description:** State the domain, without top-level domain, that should be recognized by the URL matching system. As further explained below, the assigned string is embedded into a regular expression, meaning that you may also include properly escaped regular expressions into the scheme. **Never forget to escape** characters with a particular function in regexp, such as ”.“ and ”/“. Importantly, escapes require **two** instead of just one extra backslash: one for escaping the backslash character in the JSON string, and the resulting backslash to escape the following character in the regular expression into which the scheme is embedded. **NOTE:** While there is no inbuilt technical restriction, regexp-only schemes which match to any domain will not be recognized as valid contributions, and will **not be tolerated!** A scheme must match to a specific domain, but you may allow for multiple subdomains, such as in the above stated ”Science“ example.

2. **Property:** ”top“ – **Type:** *String* or *Array* – **Required:** yes – **Description:** State the top-level domain(s) that should be recognized by the URL matching system. If a *string* is provided, and if the found top-level domain is NOT matched precisely by the provided string, the whole matching procedure terminates without returning a valid site adjuster. Note, however, that the provided string is again embedded into a regular expression, in order to improve compatibility to websites which exist under many top-level domains. Regexp-only matching is tolerated if it makes sense, i.e., if the

domain exists under many different top-level domains (see Google adjuster as an example).

If an array is provided, each element of this array must be another JSON object. This object must have a ”scheme“ property in the format described in point 1, and can furthermore have a ”path“,”prefselector“ and ”preformatter“ property, in the same format as described below in points 3, 4, and 5. In this case, the ”scheme“ string is the string specifying the top-level domain as stated above. The search for top-level domains either stops at the first array element with a positively matched scheme, or terminates the entire matching procedure without returning a site adjuster if no top-level domain scheme matches.

The purpose of providing an array with several such objects is to let the URL matcher choose different prefselectors and preformatters for different top-level domains. One example from the source where this is heavily used is the Amazon store. As the code excerpt below demonstrates, only one prefselector is necessary, but each top-level domain results in choosing a different preformatter that is adjusted to the language used on the webpage of the particular top-level domain.

```
{ "scheme": "amazon" , "prefselector": "amazon" ,
  "top": [
   { "scheme": "com" , "preformatter": "amazon-com" },
   { "scheme": "de" , "preformatter": "amazon-de" },
   { "scheme": "it" , "preformatter": "amazon-it" },
   { "scheme": "fr" , "preformatter": "amazon-fr" },
   ...
        ]
},
```

3. **Property:** ”path“ – **Type:** *String* or *Array* – **Required:** no – **Description:** State the URL path that should be recognized by the URL matching system. If a *string* is provided, the matching is only positive if the **beginning** of the found path, i.e., the part following the ”/“ after the top-level domain up to some character, is matched by the provided string. If the provided path string does not match, the entire matching procedure terminates without returning a valid site adjuster. Again, note that the path scheme may be embedded into a regular expression. The rules for contributing are the same as for the ”scheme“ property.

   Just as for the ”top“ property, assigning instead an array to ”path“ requires that each element of this array must be another JSON object. This object must have a ”scheme“ property in the format described under point 1, and can furthermore have a ”prefselector“ and ”preformatter“ property, in the same format as described below in points 4 and 5; the ”top“ property is not valid in this case. The ”scheme“ string is the string specifying the path as explained above. If such an array of paths is assigned, a positive match is only obtained if at least one provided path scheme does match in the way described above. After the first positive match, the search stops. If no scheme matches, the matching procedure terminates without return a valid site adjuster.

The purpose of assigning an array to ”path“ is the same as for ”top“: to choose different prefselectors/preformatters for different paths. An example from the code where this is used is the ”ScienceDirect“ portal, where different adjusters are applied for journal articles and books:

```
{ "scheme": "sciencedirect" , "top": "com" ,
  "path": [
           { "scheme": "science\\/book" ,
             "prefselector": "sciencedirect-book" ,
             "preformatter": "sciencedirect-book" },
           { "scheme": "science\\/article" ,
             "prefselector": "sciencedirect" ,
             "preformatter": "sciencedirect" }
         ]
},
```

4. **Property:** ”prefselector“ – **Type:** *String* – **Required:** no – **Description:** State the name of the prefselector javascript file, relative to

   $PLUGINDIR/extractors/prefselectors/

   and without the ”.js“ file extension.

5. **Property:** ”preformatter“ – **Type:** *String* – **Required:** no – **Description:** State the name of the preformatter javascript file, relative to

   $PLUGINDIR/background/preformatters/

   and without the ”.js“ file extension.

C.2 Prefselector/Preformatter preference

If any JSON object in the list specifies multiple prefselector/preformatter files in different properties, the final one is chosen according to the following preference list (1 = highest, 3 = lowest preference):

1. ”prefselector“/”preformatter“ property of an object in an array assigned to the ”path“ property. This ”path“ property can itself be either from the base object in the adjuster list, or in an object inside an array assigned to the ”top“ property.

2. ”prefselector“/”preformatter“ property of an object in an array assigned to the ”top“ property.

3. ”prefselector“/”preformatter“ property of the base object in the adjuster list.

C.3 Matching procedure

The URL is matched as follows. Prior to the matching, the protocol scheme ”http://“ or ”https://“ (actually called ”scheme“, but here to be distinguished from the above explained ”scheme“ property!) and any ”www“ subdomain following right after the protocol scheme are removed from obtained URL. The remaining part of the URL is separated into a domain, top-level domain, and a path. The domain contains all characters from the beginning up to (but not including) the last period separating the domain from the top-level domain. The top-level domain contains all characters after this period up to the first ”/“ separating the path from the domain, or up to the end of the URL if no path is included. Every character after the first ”/“ belongs to the path.

The domain-scheme specified in the adjuster list must match **the entire domain** obtained as described in the previous paragraph. To match multiple subdomains, you may use regular expressions, see previous Sec. C.1. The same as for the domain also holds for the top-level domain. For the path, the scheme obtained from the adjuster list is matched from the beginning of the path. A positive match hence only requires that the obtained path **begins** with the specified path-scheme.

D Format of prefselectors

As pointed out in section 3, the first crucial ingredient of a site adjuster are preferred selectors, prefselectors. They are entered as properties of the JSON object ”prefselectorMsg“ defined in the prefselector file:

```
var prefselectorMsg = {
...
#BIBFIELD : [#PREFSELECTOR_1, #PREFSELECTOR_2,...] ,
...
}
```

where #BIBFIELD is one of the bibliography fields described in App. A, and each preferred selector #PREFSELECTOR$_i$ itself is an array of several required and several optional arguments:

```
[
 #CSS_SELECTORS , #ATTRIBUTE_TO_READ ,
 #ALLOW_MULTIPLE_LINES , #MAXIMUM_NUMBER_OF_CHARS ,
 #ALLOW_HTML_TAGS , #MAXIMUM_NUMBER_OF_HITS
]
```

The following example of a full, valid ”prefselectorMsg“ is taken from the ”APS Journals“ site adjuster (State: February 2018):

```
var prefselectorMsg = {
 citation_issn: [ ['p.legal','innerText'] ],
 citation_download: [ ['a#export-article-link','href'] ,
                      ['BINURL','']
                    ],
 citation_abstract: [
  ['div#article-content section.abstract div.content p',
  'innerText', true, 20000] ,
  ['meta[name="description"]','content', true, 20000]
                    ],
 citation_author: [
  ['meta[name="citation_author"]','content'] ,
  ['div#title h5.authors','innerText']
                  ],
 citation_keywords: [
  ['div.physh-tagging a.physh-concept','innerText']
                    ]
};
```

D.1 Prefselector arguments

Let us describe the arguments in detail.

1. **Argument:** #CSS_SELECTORS – **Type:** *String* – **Required:** yes – **Description:** String specifying the CSS selector(s) used to select the HTML tag(s) from which the data for the #BIBFIELD should be obtained. For #BIBFIELDS from class 1 defined in App. A, only the first positive, i.e., non-empty result for the #ATTRIBUTE_TO_READ obtained by ANY of the specified selectors will be selected. In class 2, all non-empty results from all selectors will be taken into account. Examples are

   ```
   'meta[name="citation_author"]'
   'a.citLink'
   'div#title span.titleText'
   ```

2. **Argument:** #ATTRIBUTE_TO_READ – **Type:** *String* – **Required:** yes – **Description:** Attribute of the selected HTML tag(s) that should be read as the data for the #BIBFIELD. This attribute could be any standard or non-standard attribute of the selected HTML tag(s). Furthermore, you may set #ATTRIBUTE_TO_READ to either “innerText” or “textContent” in order to read the content **IN BETWEEN** the tags,

   ```
    <tag>#content</tag>
   ```

   , via the **“innerText”** or **“textContent”** property. The difference between the two properties is explained, e.g., at Stack Overflow. You can ignore all the warnings about “innerText” not being standard, too slow, or not being supported. It is supported for all browsers for which **BibItNow!** is developed, works more than fast enough for the desired performance of the plugin (=there is no loop over millions of calls of innerText), and has its legitimate purpose. In fact, most site adjusters of **BibItNow!** use

”innerText“, as it comes closer to what the publisher intends to show on its website. Examples are

```
'content'
'href'
'innerText'
```

3. **Argument:** #ALLOW_MULTIPLE_LINES – **Type:** *Boolean* – **Required:** no – **Default:** *false* – **Description:** If *false*, any data string retrieved by setting #CSS_SELECTORS and #ATTRIBUTE_TO_READ is trimmed after the first occurrence of a newline character, to avoid passing too much data to the parser! If explicitly set to *true*, any newline character is converted to a simple white space character. The above given excerpt from the ”APS Journals“ prefselector shows that a typical #BIBFIELD for which it makes sense to set this argument to *true* is the article's abstract, which may very well contain such newline characters.

4. **Argument:** #MAXIMUM_NUMBER_OF_CHARS – **Type:** *Positive Integer* – **Required:** no – **Default:** 1024 – **Description:** Sets the maximum number of characters of the string retrieved by #CSS_SELECTORS and #ATTRIBUTE_TO_READ that should be passed to the parser. By default, this number is set to 1024, in order to avoid passing too much data to the parser. The above given excerpt from the ”APS Journals“ prefselector shows that a typical #BIBFIELD for which it makes sense to set a number different from 1024 is the article's abstract, which may very well contain more characters.

5. **Argument:** #ALLOW_HTML_TAGS – **Type:** *Boolean* – **Required:** no – **Default:** *false* – **Description:** If *false*, any HTML Tag found in data string retrieved by #CSS_SELECTORS and #ATTRIBUTE_TO_READ is removed. Note that reading either the ”innerText“ or ”textContent“ property itself always involves an inbuilt HTML Tag cleanup, regardless of whether #ALLOW_HTML_TAGS is set to *true* or *false*. The cleanup performed by **BibItNow!** is an independent cleanup that is also applied when reading from attributes other than ”innerText“ or ”textContent“. The purpose of this cleanup is to minimize the risk of websites sneaking in script tags into the formatted citation data which could be malicious if they were parsed using ”eval“ or the ”innerHTML“ property.

6. **Argument:** #MAXIMUM_NUMBER_OF_HITS – **Type:** *Positive Integer* – **Required:** no – **Default:** depends on #BIBFIELD, see App. A – **Description:** Sets the maximum number of non-empty readings passed to the parser for any #BIBFIELD from class 2. For example, the number of found tags with author information is, by default, capped at 10000. In general, such upper limits are set to establish a balance between papers having a large set of valid meta information (e.g., hyper authoring with several 1000 authors) and a protection against (malicious) websites containing too many invisible meta tags.