Captured 2026-10-30T08:17:22.698Z from https://chromium.googlesource.com/chromium/src/+/refs/tags/64.0.3282.24/components/spellcheck/browser/spelling_service_client.cc

blob: 05567414e5825f01a40409c490765df6e79832cd [file]

```
1  // Copyright (c) 2012 The Chromium Authors. All rights reserved.
2  // Use of this source code is governed by a BSD-style license that can be
3  // found in the LICENSE file.
4
5  #include "components/spellcheck/browser/spelling_service_client.h"
6
7  #include <stddef.h>
8
9  #include <algorithm>
10
11 #include "base/json/json_reader.h"
12 #include "base/json/string_escape.h"
13 #include "base/logging.h"
14 #include "base/memory/ptr_util.h"
15 #include "base/stl_util.h"
16 #include "base/strings/string_util.h"
17 #include "base/strings/stringprintf.h"
18 #include "base/strings/utf_string_conversions.h"
19 #include "base/values.h"
20 #include "components/data_use_measurement/core/data_use_user_data.h"
21 #include "components/prefs/pref_service.h"
22 #include "components/spellcheck/browser/pref_names.h"
23 #include "components/spellcheck/common/spellcheck_common.h"
24 #include "components/spellcheck/common/spellcheck_result.h"
25 #include "components/user_prefs/user_prefs.h"
26 #include "content/public/browser/browser_context.h"
27 #include "content/public/browser/storage_partition.h"
28 #include "google_apis/google_api_keys.h"
29 #include "net/base/load_flags.h"
30 #include "net/url_request/url_fetcher.h"
31 #include "url/gurl.h"
32
33 namespace {
34
35 // The URL for requesting spell checking and sending user feedback.
36 const char kSpellingServiceURL[] = "https://www.googleapis.com/rpc";
37
38 // The location of spellcheck suggestions in JSON response from spelling
39 // service.
40 const char kMisspellingsPath[] = "result.spellingCheckResponse.misspellings";
```

```
41
42  // The location of error messages in JSON response from spelling service.
43  const char kErrorPath[] = "error";
44
45  // Languages currently supported by SPELLCHECK.
46  const char* const kValidLanguages[] = {"en", "es", "fi", "da"};
47
48  }  // namespace
49
50  SpellingServiceClient::SpellingServiceClient() {}
51
52  SpellingServiceClient::~SpellingServiceClient() {}
53
54  bool SpellingServiceClient::RequestTextCheck(
55      content::BrowserContext* context,
56      ServiceType type,
57      const base::string16& text,
58      const TextCheckCompleteCallback& callback) {
59    DCHECK(type == SUGGEST || type == SPELLCHECK);
60    if (!context || !IsAvailable(context, type)) {
61      callback.Run(false, text, std::vector<SpellCheckResult>());
62      return false;
63    }
64
65    const PrefService* pref = user_prefs::UserPrefs::Get(context);
66    DCHECK(pref);
67
68    std::string dictionary;
69    pref->GetList(spellcheck::prefs::kSpellCheckDictionaries)
70        ->GetString(0, &dictionary);
71
72    std::string language_code;
73    std::string country_code;
74    spellcheck::GetISOLanguageCountryCodeFromLocale(dictionary, &language_code,
75                                                    &country_code);
76
77    // Replace typographical apostrophes with typewriter apostrophes, so that
78    // server word breaker behaves correctly.
79    const base::char16 kApostrophe = 0x27;
80    const base::char16 kRightSingleQuotationMark = 0x2019;
81    base::string16 text_copy = text;
82    std::replace(text_copy.begin(), text_copy.end(), kRightSingleQuotationMark,
83                 kApostrophe);
84
85    // Format the JSON request to be sent to the Spelling service.
86    std::string encoded_text = base::GetQuotedJSONString(text_copy);
87
88    static const char kSpellingRequest[] =
89        "{"
```

```
      "\"method\":\"spelling.check\","
      "\"apiVersion\":\"v%d\","
      "\"params\":{"
      "\"text\":%s,"
      "\"language\":\"%s\","
      "\"originCountry\":\"%s\","
      "\"key\":%s"
      "}"
      "}";
  std::string api_key = base::GetQuotedJSONString(google_apis::GetAPIKey());
  std::string request = base::StringPrintf(
      kSpellingRequest, type, encoded_text.c_str(), language_code.c_str(),
      country_code.c_str(), api_key.c_str());

  GURL url = GURL(kSpellingServiceURL);

  // Create traffic annotation tag.
  net::NetworkTrafficAnnotationTag traffic_annotation =
      net::DefineNetworkTrafficAnnotation("spellcheck_lookup", R"(
        semantics {
          sender: "Online Spellcheck"
          description:
            "Chromium can provide smarter spell-checking, by sending the text "
            "that the users type into the browser, to Google's servers. This"
            "allows users to use the same spell-checking technology used by "
            "Google products, such as Docs. If the feature is enabled, "
            "Chromium will send the entire contents of text fields as user "
            "types them to Google, along with the browser's default language. "
            "Google returns a list of suggested spellings, which will be "
            "displayed in the context menu."
          trigger: "User types text into a text field or asks to correct a "
                   "misspelled word."
          data: "Text a user has typed into a text field. No user identifier "
                "is sent along with the text."
          destination: GOOGLE_OWNED_SERVICE
        }
        policy {
          cookies_allowed: NO
          setting:
            "Users can enable or disable this feature via 'Use a web service "
            "to help resolve spelling errors.' in Chromium's settings under "
            "Advanced. The feature is disabled by default."
          chrome_policy {
            SpellCheckServiceEnabled {
                policy_options {mode: MANDATORY}
                SpellCheckServiceEnabled: false
            }
          }
        })");
```

```
139
140   net::URLFetcher* fetcher =
141       CreateURLFetcher(url, traffic_annotation).release();
142   data_use_measurement::DataUseUserData::AttachToFetcher(
143       fetcher, data_use_measurement::DataUseUserData::SPELL_CHECKER);
144   fetcher->SetRequestContext(
145       content::BrowserContext::GetDefaultStoragePartition(context)
146           ->GetURLRequestContext());
147   fetcher->SetUploadData("application/json", request);
148   fetcher->SetLoadFlags(net::LOAD_DO_NOT_SEND_COOKIES |
149                         net::LOAD_DO_NOT_SAVE_COOKIES);
150   spellcheck_fetchers_[fetcher] = base::MakeUnique<TextCheckCallbackData>(
151       base::WrapUnique(fetcher), callback, text);
152   fetcher->Start();
153   return true;
154 }
155
156 bool SpellingServiceClient::IsAvailable(content::BrowserContext* context,
157                                         ServiceType type) {
158   const PrefService* pref = user_prefs::UserPrefs::Get(context);
159   DCHECK(pref);
160   // If prefs don't allow spellchecking, if the context is off the record, or if
161   // multilingual spellchecking is enabled the spelling service should be
162   // unavailable.
163   if (!pref->GetBoolean(spellcheck::prefs::kEnableSpellcheck) ||
164       !pref->GetBoolean(spellcheck::prefs::kSpellCheckUseSpellingService) ||
165       context->IsOffTheRecord())
166     return false;
167
168   // If the locale for spelling has not been set, the user has not decided to
169   // use spellcheck so we don't do anything remote (suggest or spelling).
170   std::string locale;
171   pref->GetList(spellcheck::prefs::kSpellCheckDictionaries)
172       ->GetString(0, &locale);
173   if (locale.empty())
174     return false;
175
176   // Finally, if all options are available, we only enable only SUGGEST
177   // if SPELLCHECK is not available for our language because SPELLCHECK results
178   // are a superset of SUGGEST results.
179   for (const char* language : kValidLanguages) {
180     if (!locale.compare(0, 2, language))
181       return type == SPELLCHECK;
182   }
183
184   // Only SUGGEST is allowed.
185   return type == SUGGEST;
186 }
187
```

```
188 bool SpellingServiceClient::ParseResponse(
189     const std::string& data,
190     std::vector<SpellCheckResult>* results) {
191   // When this JSON-RPC call finishes successfully, the Spelling service returns
192   // an JSON object listed below.
193   // * result - an envelope object representing the result from the APIARY
194   //   server, which is the JSON-API front-end for the Spelling service. This
195   //   object consists of the following variable:
196   //   - spellingCheckResponse (SpellingCheckResponse).
197   // * SpellingCheckResponse - an object representing the result from the
198   //   Spelling service. This object consists of the following variable:
199   //   - misspellings (optional array of Misspelling)
200   // * Misspelling - an object representing a misspelling region and its
201   //   suggestions. This object consists of the following variables:
202   //   - charStart (number) - the beginning of the misspelled region;
203   //   - charLength (number) - the length of the misspelled region;
204   //   - suggestions (array of string) - the suggestions for the misspelling
205   //     text, and;
206   //   - canAutoCorrect (optional boolean) - whether we can use the first
207   //     suggestion for auto-correction.
208   // For example, the Spelling service returns the following JSON when we send a
209   // spelling request for "duck goes quisk" as of 16 August, 2011.
210   // {
211   //   "result": {
212   //     "spellingCheckResponse": {
213   //       "misspellings": [{
214   //           "charStart": 10,
215   //           "charLength": 5,
216   //           "suggestions": [{ "suggestion": "quack" }],
217   //           "canAutoCorrect": false
218   //       }]
219   //     }
220   //   }
221   // }
222   // If the service is not available, the Spelling service returns JSON with an
223   // error.
224   // {
225   //   "error": {
226   //     "code": 400,
227   //     "message": "Bad Request",
228   //     "data": [...]
229   //   }
230   // }
231   std::unique_ptr<base::DictionaryValue> value(
232       static_cast<base::DictionaryValue*>(
233           base::JSONReader::Read(data, base::JSON_ALLOW_TRAILING_COMMAS)
234               .release()));
235   if (!value.get() || !value->is_dict())
236     return false;
```

```
237
238   // Check for errors from spelling service.
239   base::DictionaryValue* error = nullptr;
240   if (value->GetDictionary(kErrorPath, &error))
241     return false;
242
243   // Retrieve the array of Misspelling objects. When the input text does not
244   // have misspelled words, it returns an empty JSON. (In this case, its HTTP
245   // status is 200.) We just return true for this case.
246   base::ListValue* misspellings = nullptr;
247   if (!value->GetList(kMisspellingsPath, &misspellings))
248     return true;
249
250   for (size_t i = 0; i < misspellings->GetSize(); ++i) {
251     // Retrieve the i-th misspelling region and put it to the given vector. When
252     // the Spelling service sends two or more suggestions, we read only the
253     // first one because SpellCheckResult can store only one suggestion.
254     base::DictionaryValue* misspelling = nullptr;
255     if (!misspellings->GetDictionary(i, &misspelling))
256       return false;
257
258     int start = 0;
259     int length = 0;
260     base::ListValue* suggestions = nullptr;
261     if (!misspelling->GetInteger("charStart", &start) ||
262         !misspelling->GetInteger("charLength", &length) ||
263         !misspelling->GetList("suggestions", &suggestions)) {
264       return false;
265     }
266
267     base::DictionaryValue* suggestion = nullptr;
268     base::string16 replacement;
269     if (!suggestions->GetDictionary(0, &suggestion) ||
270         !suggestion->GetString("suggestion", &replacement)) {
271       return false;
272     }
273     SpellCheckResult result(SpellCheckResult::SPELLING, start, length,
274                             replacement);
275     results->push_back(result);
276   }
277   return true;
278 }
279
280 SpellingServiceClient::TextCheckCallbackData::TextCheckCallbackData(
281     std::unique_ptr<net::URLFetcher> fetcher,
282     TextCheckCompleteCallback callback,
283     base::string16 text)
284     : fetcher(std::move(fetcher)), callback(callback), text(text) {}
285
```

```
286 SpellingServiceClient::TextCheckCallbackData::~TextCheckCallbackData() {}
287
288 void SpellingServiceClient::OnURLFetchComplete(const net::URLFetcher* source) {
289   DCHECK(base::ContainsKey(spellcheck_fetchers_, source));
290   std::unique_ptr<TextCheckCallbackData> callback_data =
291       std::move(spellcheck_fetchers_[source]);
292   spellcheck_fetchers_.erase(source);
293
294   bool success = false;
295   std::vector<SpellCheckResult> results;
296   if (source->GetResponseCode() / 100 == 2) {
297     std::string data;
298     source->GetResponseAsString(&data);
299     success = ParseResponse(data, &results);
300   }
301
302   // The callback may release the last (transitive) dependency on |this|. It
303   // MUST be the last function called.
304   callback_data->callback.Run(success, callback_data->text, results);
305 }
306
307 std::unique_ptr<net::URLFetcher> SpellingServiceClient::CreateURLFetcher(
308     const GURL& url,
309     net::NetworkTrafficAnnotationTag traffic_annotation) {
310   return net::URLFetcher::Create(url, net::URLFetcher::POST, this,
311                                  traffic_annotation);
312 }
```

Powered by Gitiles | Privacy | Terms txt json