

834e3eabdd

...

drogon / lib / src / StaticFileRouter.cc

Bertrand Darbon Add option to set default handler (#802) ✓

History

4 contributors

452 lines (437 sloc) 15.4 KB

```

1  /**
2  *
3  * StaticFileRouter.cc
4  * An Tao
5  *
6  * Copyright 2018, An Tao. All rights reserved.
7  * https://github.com/an-tao/drogon
8  * Use of this source code is governed by a MIT license
9  * that can be found in the License file.
10 *
11 * Drogon
12 *
13 */
14
15 #include "StaticFileRouter.h"
16 #include "HttpAppFrameworkImpl.h"
17 #include "HttpRequestImpl.h"
18 #include "HttpResponseImpl.h"
19 #include <fstream>
20 #include <iostream>
21 #include <algorithm>
22 #include <fcntl.h>
23 #ifndef _WIN32
24 #include <sys/file.h>
25 #else
26 #define stat _stati64
27 #define S_ISREG(m) (((m)&0170000) == (0100000))
28 #define S_ISDIR(m) (((m)&0170000) == (0040000))
29 #endif
30 #include <sys/stat.h>
31
32 using namespace drogon;
33
34 void StaticFileRouter::init(const std::vector<trantor::EventLoop*> &ioloops)
35 {
36     // Max timeout up to about 70 days;
37     staticFilesCacheMap_ = decltype(staticFilesCacheMap_)(
38         new IOThreadStorage<std::unique_ptr<CacheMap<std::string, char>>>());
39     staticFilesCacheMap_->init(
40         [&ioloops](std::unique_ptr<CacheMap<std::string, char>> &mapPtr,
41             size_t i) {
42             assert(i == ioloops[i]->index());
43             mapPtr = std::unique_ptr<CacheMap<std::string, char>>{
44                 new CacheMap<std::string, char>(ioloops[i], 1.0, 4, 50)};
45             });
46     staticFilesCache_ = decltype(staticFilesCache_)(
47         new IOThreadStorage<
48             std::unordered_map<std::string, HttpResponsePtr>>());
49     ioLocationsPtr_ =
50         decltype(ioLocationsPtr_)(new IOThreadStorage<std::vector<Location>>());
51     for (auto *loop : ioloops)
52     {
53         loop->queueInLoop([this] { **ioLocationsPtr_ = locations_; });
54     }
55 }
56
57 void StaticFileRouter::route(
58     const HttpRequestImplPtr &req,
59     std::function<void(const HttpResponsePtr &);> &&callback)
60 {
61     const std::string &path = req->path();
62     if (path.find("../") != std::string::npos)
63     {
64         // Downloading files from the parent folder is forbidden.
65         callback(app().getCustomErrorHandler()(k403Forbidden));
66         return;
67     }
68
69     auto lPath = path;
70     std::transform(lPath.begin(), lPath.end(), lPath.begin(), tolower);
71
72     for (auto &location : **ioLocationsPtr_)
73     {
74         auto &URI = location.uriPrefix_;
75         if (location.reallocation_.empty())
76         {
77             if (!location.alias_.empty())
78             {

```

```

79     if (location.alias_[0] == '/')
80     {
81         location.realLocation_ = location.alias_;
82     }
83     else
84     {
85         location.realLocation_ =
86             HttpAppFrameworkImpl::instance().getDocumentRoot() +
87             location.alias_;
88     }
89 }
90 else
91 {
92     location.realLocation_ =
93         HttpAppFrameworkImpl::instance().getDocumentRoot() +
94         location.uriPrefix_;
95 }
96 if (location.realLocation_[location.realLocation_.length() - 1] !=
97     '/')
98 {
99     location.realLocation_.append(1, '/');
100 }
101 if (!location.isCaseSensitive_)
102 {
103     std::transform(URI.begin(), URI.end(), URI.begin(), tolower);
104 }
105 }
106 auto &tmpPath = location.isCaseSensitive_ ? path : lPath;
107 if (tmpPath.length() >= URI.length()) &&
108     std::equal(tmpPath.begin(),
109         tmpPath.begin() + URI.length(),
110         URI.begin()))
111 {
112     string_view restOfThePath{path.data() + URI.length(),
113         path.length() - URI.length()};
114     auto pos = restOfThePath.rfind('/');
115     if (pos != 0 && pos != string_view::npos && !location.isRecursive_)
116     {
117         callback(app().getCustomErrorHandler()(k403Forbidden));
118         return;
119     }
120     std::string filePath =
121         location.realLocation_ +
122         std::string{restOfThePath.data(), restOfThePath.length()};
123     struct stat fileStat;
124     if (stat(filePath.c_str(), &fileStat) != 0)
125     {
126         defaultHandler_(req, std::move(callback));
127         return;
128     }
129     if (S_ISDIR(fileStat.st_mode))
130     {
131         // Check if path is eligible for an implicit index.html
132         if (implicitPageEnable_)
133         {
134             filePath = filePath + "/" + implicitPage_;
135         }
136         else
137         {
138             callback(app().getCustomErrorHandler()(k403Forbidden));
139             return;
140         }
141     }
142     else
143     {
144         if (!location.allowAll_)
145         {
146             pos = restOfThePath.rfind('.');
147             if (pos == string_view::npos)
148             {
149                 callback(app().getCustomErrorHandler()(k403Forbidden));
150                 return;
151             }
152             std::string extension{restOfThePath.data() + pos + 1,
153                 restOfThePath.length() - pos - 1};
154             std::transform(extension.begin(),
155                 extension.end(),
156                 extension.begin(),
157                 tolower);
158             if (fileTypeSet_.find(extension) == fileTypeSet_.end())
159             {
160                 callback(app().getCustomErrorHandler()(k403Forbidden));
161                 return;
162             }
163         }
164     }
165 }
166 if (location.filters_.empty())
167 {
168     sendStaticFileResponse(filePath,
169         req,
170         std::move(callback),
171         string_view{
172             location.defaultContentType_});
173 }
174 else
175 {
176     auto callbackPtr = std::make_shared<

```

```

177         std::function<void(const drogon::HttpResponsePtr &>>(<br>
178             std::move(callback));<br>
179         filters_function::doFilters(<br>
180             location.filters_,<br>
181             req,<br>
182             callbackPtr,<br>
183             [callbackPtr,<br>
184                 this,<br>
185                 req,<br>
186                 filePath = std::move(filePath),<br>
187                 &contentType = location.defaultContentType_]() {<br>
188             sendStaticFileResponse(filePath,<br>
189                 req,<br>
190                 std::move(*callbackPtr),<br>
191                 string_view(contentType));<br>
192         });<br>
193     }<br>
194<br>
195     return;<br>
196 }<br>
197 }<br>
198<br>
199 std::string directoryPath =<br>
200     HttpAppFrameworkImpl::instance().getDocumentRoot() + path;<br>
201 struct stat fileStat;<br>
202 if (stat(directoryPath.c_str(), &fileStat) == 0)<br>
203 {<br>
204     if (S_ISDIR(fileStat.st_mode))<br>
205     {<br>
206         // Check if path is eligible for an implicit index.html<br>
207         if (implicitPageEnable_)<br>
208         {<br>
209             std::string filePath = directoryPath + "/" + implicitPage_;<br>
210             sendStaticFileResponse(filePath, req, std::move(callback), "");<br>
211             return;<br>
212         }<br>
213         else<br>
214         {<br>
215             callback(app().getCustomErrorHandler()(k403Forbidden));<br>
216             return;<br>
217         }<br>
218     }<br>
219     else<br>
220     {<br>
221         // This is a normal page<br>
222         auto pos = path.rfind('.');<br>
223         if (pos == std::string::npos)<br>
224         {<br>
225             callback(app().getCustomErrorHandler()(k403Forbidden));<br>
226             return;<br>
227         }<br>
228         std::string filetype = lPath.substr(pos + 1);<br>
229         if (fileTypeSet_.find(filetype) != fileTypeSet_.end())<br>
230         {<br>
231             // LOG_INFO << "file query!" << path;<br>
232             std::string filePath = directoryPath;<br>
233             sendStaticFileResponse(filePath, req, std::move(callback), "");<br>
234             return;<br>
235         }<br>
236     }<br>
237 }<br>
238 defaultHandler_(req, std::move(callback));<br>
239 }<br>
240<br>
241 void StaticFileRouter::sendStaticFileResponse(<br>
242     const std::string &filePath,<br>
243     const HttpRequestImplPtr &req,<br>
244     std::function<void(const HttpResponsePtr &> &&callback,<br>
245     const string_view &defaultContentType)<br>
246 { // find cached response<br>
247     HttpResponsePtr cachedResp;<br>
248     auto &cacheMap = staticFilesCache->getThreadData();<br>
249     auto iter = cacheMap.find(filePath);<br>
250     if (iter != cacheMap.end())<br>
251     {<br>
252         cachedResp = iter->second;<br>
253     }<br>
254<br>
255     // check last modified time,rfc2616-14.25<br>
256     // If-Modified-Since: Mon, 15 Oct 2018 06:26:33 GMT<br>
257<br>
258     std::string timeStr;<br>
259     bool fileExists{false};<br>
260     if (enableLastModify_)<br>
261     {<br>
262         if (cachedResp)<br>
263         {<br>
264             if (req->method() != Get)<br>
265             {<br>
266                 callback(app().getCustomErrorHandler()(k405MethodNotAllowed));<br>
267                 return;<br>
268             }<br>
269             if (static_cast<HttpResponseImpl*>(cachedResp.get())<br>
270                 ->getHeaderBy("last-modified") ==<br>
271                 req->getHeaderBy("if-modified-since"))<br>
272             {<br>
273                 std::shared_ptr<HttpResponseImpl> resp =<br>
274                 std::make_shared<HttpResponseImpl>();

```

```

275     resp->setStatusCode(k304NotModified);
276     resp->setContentTypeCode(CT_NONE);
277     HttpAppFrameworkImpl::instance().callCallback(req,
278                                                    resp,
279                                                    callback);
280     return;
281 }
282 }
283 else
284 {
285     struct stat fileStat;
286     LOG_TRACE << "enabled LastModify";
287     if (stat(filePath.c_str(), &fileStat) == 0 &&
288         S_ISREG(fileStat.st_mode))
289     {
290         fileExists = true;
291         LOG_TRACE << "last modify time:" << fileStat.st_mtime;
292         if (req->method() != Get)
293         {
294             callback(
295                 app().getCustomErrorHandler()(k405MethodNotAllowed));
296             return;
297         }
298         struct tm tm1;
299 #ifdef _WIN32
300         gmtime_s(&tm1, &fileStat.st_mtime);
301 #else
302         gmtime_r(&fileStat.st_mtime, &tm1);
303 #endif
304         timeStr.resize(64);
305         auto len = strftime((char *)timeStr.data(),
306                             timeStr.size(),
307                             "%a, %d %b %Y %H:%M:%S GMT",
308                             &tm1);
309         timeStr.resize(len);
310         const std::string &modiStr =
311             req->getHeaderBy("if-modified-since");
312         if (modiStr == timeStr && !modiStr.empty())
313         {
314             LOG_TRACE << "not Modified!";
315             std::shared_ptr<HttpResponseImpl> resp =
316                 std::make_shared<HttpResponseImpl>();
317             resp->setStatusCode(k304NotModified);
318             resp->setContentTypeCode(CT_NONE);
319             HttpAppFrameworkImpl::instance().callCallback(req,
320                                                            resp,
321                                                            callback);
322             return;
323         }
324     }
325     else
326     {
327         defaultHandler_(req, std::move(callback));
328         return;
329     }
330 }
331 }
332 if (cachedResp)
333 {
334     if (req->method() != Get)
335     {
336         callback(app().getCustomErrorHandler()(k405MethodNotAllowed));
337         return;
338     }
339     LOG_TRACE << "Using file cache";
340     HttpAppFrameworkImpl::instance().callCallback(req,
341                                                    cachedResp,
342                                                    callback);
343     return;
344 }
345 if (!fileExists)
346 {
347     struct stat fileStat;
348     if (stat(filePath.c_str(), &fileStat) != 0 ||
349         !S_ISREG(fileStat.st_mode))
350     {
351         defaultHandler_(req, std::move(callback));
352         return;
353     }
354 }
355
356 if (req->method() != Get)
357 {
358     callback(app().getCustomErrorHandler()(k405MethodNotAllowed));
359     return;
360 }
361
362 HttpResponsePtr resp;
363 auto &acceptEncoding = req->getHeaderBy("accept-encoding");
364
365 if (brStaticFlag_ && acceptEncoding.find("br") != std::string::npos)
366 {
367     // Find compressed file first.
368     auto brFileName = filePath + ".br";
369     struct stat filestat;
370     if (stat(brFileName.c_str(), &filestat) == 0 &&
371         S_ISREG(filestat.st_mode))
372     {

```

```

373         resp =
374             HttpResponse::newFileResponse(brFileName,
375                                     "",
376                                     drogon::getContentType(filePath));
377         resp->addHeader("Content-Encoding", "br");
378     }
379 }
380 if (!resp && gzipStaticFlag_ &&
381     acceptEncoding.find("gzip") != std::string::npos)
382 {
383     // Find compressed file first.
384     auto gzipFileName = filePath + ".gz";
385     struct stat filestat;
386     if (stat(gzipFileName.c_str(), &filestat) == 0 &&
387         S_ISREG(filestat.st_mode))
388     {
389         resp =
390             HttpResponse::newFileResponse(gzipFileName,
391                                     "",
392                                     drogon::getContentType(filePath));
393         resp->addHeader("Content-Encoding", "gzip");
394     }
395 }
396 if (!resp)
397     resp = HttpResponse::newFileResponse(filePath);
398 if (resp->statusCode() != k404NotFound)
399 {
400     if (resp->getContentType() == CT_APPLICATION_OCTET_STREAM &&
401         !defaultContentType.empty())
402     {
403         resp->setContentTypeCodeAndCustomString(CT_CUSTOM,
404                                             defaultContentType);
405     }
406     if (!timeStr.empty())
407     {
408         resp->addHeader("Last-Modified", timeStr);
409         resp->addHeader("Expires", "Thu, 01 Jan 1970 00:00:00 GMT");
410     }
411     if (!headers_.empty())
412     {
413         for (auto &header : headers_)
414         {
415             resp->addHeader(header.first, header.second);
416         }
417     }
418     // cache the response for 5 seconds by default
419     if (staticFilesCacheTime_ >= 0)
420     {
421         LOG_TRACE << "Save in cache for " << staticFilesCacheTime_
422             << " seconds";
423         resp->setExpiredTime(staticFilesCacheTime_);
424         staticFilesCache_->getThreadData()[filePath] = resp;
425         staticFilesCacheMap_->getThreadData()->insert(
426             filePath, 0, staticFilesCacheTime_, [this, filePath]() {
427                 LOG_TRACE << "Erase cache";
428                 assert(staticFilesCache_->getThreadData().find(filePath) !=
429                     staticFilesCache_->getThreadData().end());
430                 staticFilesCache_->getThreadData().erase(filePath);
431             });
432     }
433     HttpAppFrameworkImpl::instance().callCallback(req, resp, callback);
434     return;
435 }
436 callback(resp);
437 return;
438 }
439 void StaticFileRouter::setFileTypes(const std::vector<std::string> &types)
440 {
441     fileTypeSet_.clear();
442     for (auto const &type : types)
443     {
444         fileTypeSet_.insert(type);
445     }
446 }
447 void StaticFileRouter::defaultHandler(
448     const HttpRequestPtr & /*req*/,
449     std::function<void(const HttpResponsePtr &)> &&callback)
450 {
451     callback(HttpResponse::newNotFoundResponse());
452 }

```