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

4 contributors

452 lines (437 sloc) | 15.4 KB

```
/**
 *
 *  StaticFileRouter.cc
 *  An Tao
 *
 *  Copyright 2018, An Tao.  All rights reserved.
 *  https://github.com/an-tao/drogon
 *  Use of this source code is governed by a MIT license
 *  that can be found in the License file.
 *
 *  Drogon
 *
 */

#include "StaticFileRouter.h"
#include "HttpAppFrameworkImpl.h"
#include "HttpRequestImpl.h"
#include "HttpResponseImpl.h"
#include <fstream>
#include <iostream>
#include <algorithm>
#include <fcntl.h>
#ifndef _WIN32
#include <sys/file.h>
#else
#define stat _stati64
#define S_ISREG(m) (((m)&0170000) == (0100000))
#define S_ISDIR(m) (((m)&0170000) == (0040000))
#endif
#include <sys/stat.h>

using namespace drogon;

void StaticFileRouter::init(const std::vector<trantor::EventLoop *> &ioloops)
{
    // Max timeout up to about 70 days;
    staticFilesCacheMap_ = decltype(staticFilesCacheMap_)(
        new IOThreadStorage<std::unique_ptr<CacheMap<std::string, char>>>);
    staticFilesCacheMap_->init(
        [&ioloops](std::unique_ptr<CacheMap<std::string, char>> &mapPtr,
                   size_t i) {
            assert(i == ioloops[i]->index());
            mapPtr = std::unique_ptr<CacheMap<std::string, char>>(
                new CacheMap<std::string, char>(ioloops[i], 1.0, 4, 50));
        });
    staticFilesCache_ = decltype(staticFilesCache_)(
        new IOThreadStorage<
            std::unordered_map<std::string, HttpResponsePtr>>{});
    ioLocationsPtr_ =
        decltype(ioLocationsPtr_)(new IOThreadStorage<std::vector<Location>>);
    for (auto *loop : ioloops)
    {
        loop->queueInLoop([this] { **ioLocationsPtr_ = locations_; });
    }
}

void StaticFileRouter::route(
    const HttpRequestImplPtr &req,
    std::function<void(const HttpResponsePtr &)> &&callback)
{
    const std::string &path = req->path();
    if (path.find("/../") != std::string::npos)
    {
        // Downloading files from the parent folder is forbidden.
        callback(app().getCustomErrorHandler()(k403Forbidden));
        return;
    }

    auto lPath = path;
    std::transform(lPath.begin(), lPath.end(), lPath.begin(), tolower);

    for (auto &location : **ioLocationsPtr_)
    {
        auto &URI = location.uriPrefix_;
        if (location.realLocation_.empty())
        {
            if (!location.alias_.empty())
            {
```

```
            if (location.alias_[0] == '/')
            {
                location.realLocation_ = location.alias_;
            }
            else
            {
                location.realLocation_ =
                    HttpAppFrameworkImpl::instance().getDocumentRoot() +
                    location.alias_;
            }
        }
        else
        {
            location.realLocation_ =
                HttpAppFrameworkImpl::instance().getDocumentRoot() +
                location.uriPrefix_;
        }
        if (location.realLocation_[location.realLocation_.length() - 1] !=
            '/')
        {
            location.realLocation_.append(1, '/');
        }
        if (!location.isCaseSensitive_)
        {
            std::transform(URI.begin(), URI.end(), URI.begin(), tolower);
        }
    }
    auto &tmpPath = location.isCaseSensitive_ ? path : lPath;
    if (tmpPath.length() >= URI.length() &&
        std::equal(tmpPath.begin(),
                   tmpPath.begin() + URI.length(),
                   URI.begin()))
    {
        string_view restOfThePath{path.data() + URI.length(),
                                  path.length() - URI.length()};
        auto pos = restOfThePath.rfind('/');
        if (pos != 0 && pos != string_view::npos && !location.isRecursive_)
        {
            callback(app().getCustomErrorHandler()(k403Forbidden));
            return;
        }
        std::string filePath =
            location.realLocation_ +
            std::string{restOfThePath.data(), restOfThePath.length()};
        struct stat fileStat;
        if (stat(filePath.c_str(), &fileStat) != 0)
        {
            defaultHandler_(req, std::move(callback));
            return;
        }
        if (S_ISDIR(fileStat.st_mode))
        {
            // Check if path is eligible for an implicit index.html
            if (implicitPageEnable_)
            {
                filePath = filePath + "/" + implicitPage_;
            }
            else
            {
                callback(app().getCustomErrorHandler()(k403Forbidden));
                return;
            }
        }
        else
        {
            if (!location.allowAll_)
            {
                pos = restOfThePath.rfind('.');
                if (pos == string_view::npos)
                {
                    callback(app().getCustomErrorHandler()(k403Forbidden));
                    return;
                }
                std::string extension{restOfThePath.data() + pos + 1,
                                      restOfThePath.length() - pos - 1};
                std::transform(extension.begin(),
                               extension.end(),
                               extension.begin(),
                               tolower);
                if (fileTypeSet_.find(extension) == fileTypeSet_.end())
                {
                    callback(app().getCustomErrorHandler()(k403Forbidden));
                    return;
                }
            }
        }

        if (location.filters_.empty())
        {
            sendStaticFileResponse(filePath,
                                   req,
                                   std::move(callback),
                                   string_view{
                                       location.defaultContentType_});
        }
        else
        {
            auto callbackPtr = std::make_shared<
```

```
                    std::function<void(const drogon::HttpResponsePtr &)>>(
                    std::move(callback));
                filters_function::doFilters(
                    location.filters_,
                    req,
                    callbackPtr,
                    [callbackPtr,
                     this,
                     req,
                     filePath = std::move(filePath),
                     &contentType = location.defaultContentType_]() {
                        sendStaticFileResponse(filePath,
                                               req,
                                               std::move(*callbackPtr),
                                               string_view{contentType});
                    });
            }

            return;
        }
    }

    std::string directoryPath =
        HttpAppFrameworkImpl::instance().getDocumentRoot() + path;
    struct stat fileStat;
    if (stat(directoryPath.c_str(), &fileStat) == 0)
    {
        if (S_ISDIR(fileStat.st_mode))
        {
            // Check if path is eligible for an implicit index.html
            if (implicitPageEnable_)
            {
                std::string filePath = directoryPath + "/" + implicitPage_;
                sendStaticFileResponse(filePath, req, std::move(callback), "");
                return;
            }
            else
            {
                callback(app().getCustomErrorHandler()(k403Forbidden));
                return;
            }
        }
        else
        {
            // This is a normal page
            auto pos = path.rfind('.');
            if (pos == std::string::npos)
            {
                callback(app().getCustomErrorHandler()(k403Forbidden));
                return;
            }
            std::string filetype = lPath.substr(pos + 1);
            if (fileTypeSet_.find(filetype) != fileTypeSet_.end())
            {
                // LOG_INFO << "file query!" << path;
                std::string filePath = directoryPath;
                sendStaticFileResponse(filePath, req, std::move(callback), "");
                return;
            }
        }
    }
    defaultHandler_(req, std::move(callback));
}

void StaticFileRouter::sendStaticFileResponse(
    const std::string &filePath,
    const HttpRequestImplPtr &req,
    std::function<void(const HttpResponsePtr &)> &&callback,
    const string_view &defaultContentType)
{  // find cached response
    HttpResponsePtr cachedResp;
    auto &cacheMap = staticFilesCache_->getThreadData();
    auto iter = cacheMap.find(filePath);
    if (iter != cacheMap.end())
    {
        cachedResp = iter->second;
    }

    // check last modified time,rfc2616-14.25
    // If-Modified-Since: Mon, 15 Oct 2018 06:26:33 GMT

    std::string timeStr;
    bool fileExists{false};
    if (enableLastModify_)
    {
        if (cachedResp)
        {
            if (req->method() != Get)
            {
                callback(app().getCustomErrorHandler()(k405MethodNotAllowed));
                return;
            }
            if (static_cast<HttpResponseImpl *>(cachedResp.get())
                    ->getHeaderBy("last-modified") ==
                req->getHeaderBy("if-modified-since"))
            {
                std::shared_ptr<HttpResponseImpl> resp =
                    std::make_shared<HttpResponseImpl>();
```

```
                resp->setStatusCode(k304NotModified);
                resp->setContentTypeCode(CT_NONE);
                HttpAppFrameworkImpl::instance().callCallback(req,
                                                              resp,
                                                              callback);
                return;
            }
        }
        else
        {
            struct stat fileStat;
            LOG_TRACE << "enabled LastModify";
            if (stat(filePath.c_str(), &fileStat) == 0 &&
                S_ISREG(fileStat.st_mode))
            {
                fileExists = true;
                LOG_TRACE << "last modify time:" << fileStat.st_mtime;
                if (req->method() != Get)
                {
                    callback(
                        app().getCustomErrorHandler()(k405MethodNotAllowed));
                    return;
                }
                struct tm tm1;
#ifdef _WIN32
                gmtime_s(&tm1, &fileStat.st_mtime);
#else
                gmtime_r(&fileStat.st_mtime, &tm1);
#endif
                timeStr.resize(64);
                auto len = strftime((char *)timeStr.data(),
                                    timeStr.size(),
                                    "%a, %d %b %Y %H:%M:%S GMT",
                                    &tm1);
                timeStr.resize(len);
                const std::string &modiStr =
                    req->getHeaderBy("if-modified-since");
                if (modiStr == timeStr && !modiStr.empty())
                {
                    LOG_TRACE << "not Modified!";
                    std::shared_ptr<HttpResponseImpl> resp =
                        std::make_shared<HttpResponseImpl>();
                    resp->setStatusCode(k304NotModified);
                    resp->setContentTypeCode(CT_NONE);
                    HttpAppFrameworkImpl::instance().callCallback(req,
                                                                  resp,
                                                                  callback);
                    return;
                }
            }
            else
            {
                defaultHandler_(req, std::move(callback));
                return;
            }
        }
    }
    if (cachedResp)
    {
        if (req->method() != Get)
        {
            callback(app().getCustomErrorHandler()(k405MethodNotAllowed));
            return;
        }
        LOG_TRACE << "Using file cache";
        HttpAppFrameworkImpl::instance().callCallback(req,
                                                      cachedResp,
                                                      callback);
        return;
    }
    if (!fileExists)
    {
        struct stat fileStat;
        if (stat(filePath.c_str(), &fileStat) != 0 ||
            !S_ISREG(fileStat.st_mode))
        {
            defaultHandler_(req, std::move(callback));
            return;
        }
    }

    if (req->method() != Get)
    {
        callback(app().getCustomErrorHandler()(k405MethodNotAllowed));
        return;
    }

    HttpResponsePtr resp;
    auto &acceptEncoding = req->getHeaderBy("accept-encoding");

    if (brStaticFlag_ && acceptEncoding.find("br") != std::string::npos)
    {
        // Find compressed file first.
        auto brFileName = filePath + ".br";
        struct stat filestat;
        if (stat(brFileName.c_str(), &filestat) == 0 &&
            S_ISREG(filestat.st_mode))
        {
```

```
            resp =
                HttpResponse::newFileResponse(brFileName,
                                              "",
                                              drogon::getContentType(filePath));
            resp->addHeader("Content-Encoding", "br");
        }
    }
    if (!resp && gzipStaticFlag_ &&
        acceptEncoding.find("gzip") != std::string::npos)
    {
        // Find compressed file first.
        auto gzipFileName = filePath + ".gz";
        struct stat filestat;
        if (stat(gzipFileName.c_str(), &filestat) == 0 &&
            S_ISREG(filestat.st_mode))
        {
            resp =
                HttpResponse::newFileResponse(gzipFileName,
                                              "",
                                              drogon::getContentType(filePath));
            resp->addHeader("Content-Encoding", "gzip");
        }
    }
    if (!resp)
        resp = HttpResponse::newFileResponse(filePath);
    if (resp->statusCode() != k404NotFound)
    {
        if (resp->getContentType() == CT_APPLICATION_OCTET_STREAM &&
            !defaultContentType.empty())
        {
            resp->setContentTypeCodeAndCustomString(CT_CUSTOM,
                                                    defaultContentType);
        }
        if (!timeStr.empty())
        {
            resp->addHeader("Last-Modified", timeStr);
            resp->addHeader("Expires", "Thu, 01 Jan 1970 00:00:00 GMT");
        }
        if (!headers_.empty())
        {
            for (auto &header : headers_)
            {
                resp->addHeader(header.first, header.second);
            }
        }
        // cache the response for 5 seconds by default
        if (staticFilesCacheTime_ >= 0)
        {
            LOG_TRACE << "Save in cache for " << staticFilesCacheTime_
                      << " seconds";
            resp->setExpiredTime(staticFilesCacheTime_);
            staticFilesCache_->getThreadData()[filePath] = resp;
            staticFilesCacheMap_->getThreadData()->insert(
                filePath, 0, staticFilesCacheTime_, [this, filePath]() {
                    LOG_TRACE << "Erase cache";
                    assert(staticFilesCache_->getThreadData().find(filePath) !=
                           staticFilesCache_->getThreadData().end());
                    staticFilesCache_->getThreadData().erase(filePath);
                });
        }
        HttpAppFrameworkImpl::instance().callCallback(req, resp, callback);
        return;
    }
    callback(resp);
    return;
}
void StaticFileRouter::setFileTypes(const std::vector<std::string> &types)
{
    fileTypeSet_.clear();
    for (auto const &type : types)
    {
        fileTypeSet_.insert(type);
    }
}
void StaticFileRouter::defaultHandler(
    const HttpRequestPtr & /*req*/,
    std::function<void(const HttpResponsePtr &)> &&callback)
{
    callback(HttpResponse::newNotFoundResponse());
}
```