Додаток А

ПРЕЗЕНТАЦІЯ

Атестаційна робота на тему:

Модель багатоплатформового Digital Signage плеєра

Виконав:
Іванов М.Ю.,
студент II курсу,
групи СКСм-20-1

Керівник:
Свірь І.Ю.
проф. каф. АПОТ ХНУРЕ

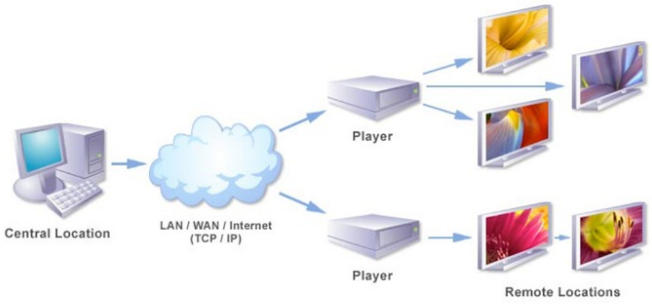
Мета роботи: розробити багатоплатформовий плеєр для відображення відео, аудіо та веб-контенту в рамках інфраструктури «Digital Signage».

Постановка задачі

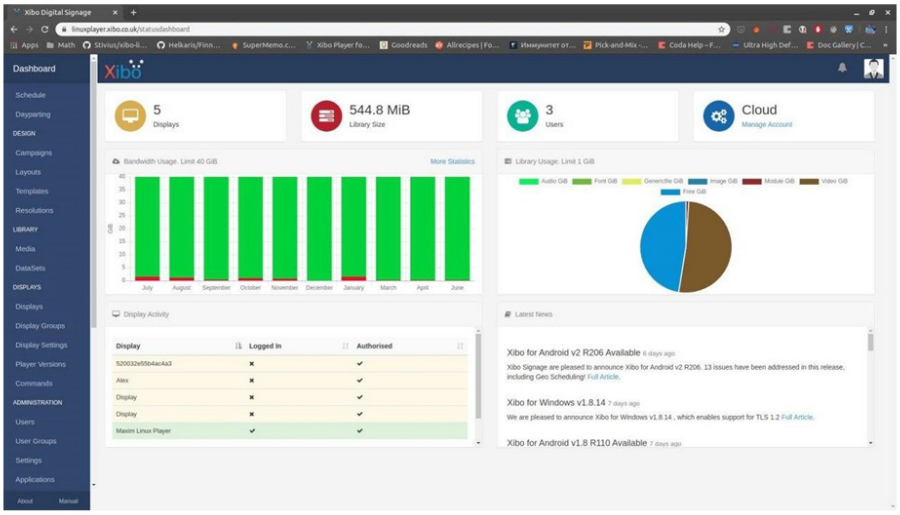
- Виконати дослідження щодо CMS, яка задовольняє нашим потребам
- Виконати дослідження апаратного прискорення на процесорі сімейства ARM
- Знайти необхідні інструменти для реалізації графічного інтерфейса та обробки аудіо- і відеоконтента
- Розробити мережеву архітектуру для плеєра
- Розробити архітектури обробки медіаконтенту
- Протестувати отримане ПЗ на кінцевому пристрої

Структура та концепція системи «Digital Sigange»

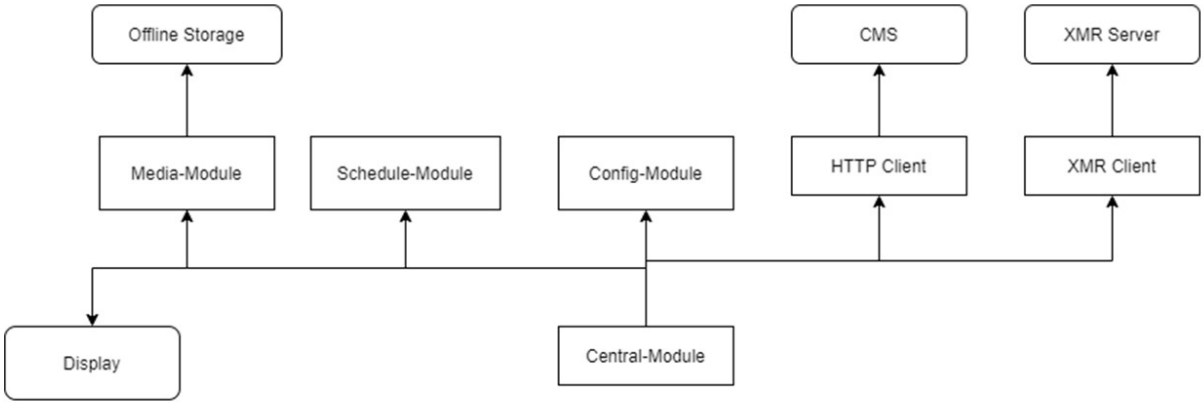
- **Digital signage** — це форма зовнішньої реклами, в якій контент та повідомлення відображаються на електронних екранах з метою донесення цільового повідомлення у певне місце та певний час.



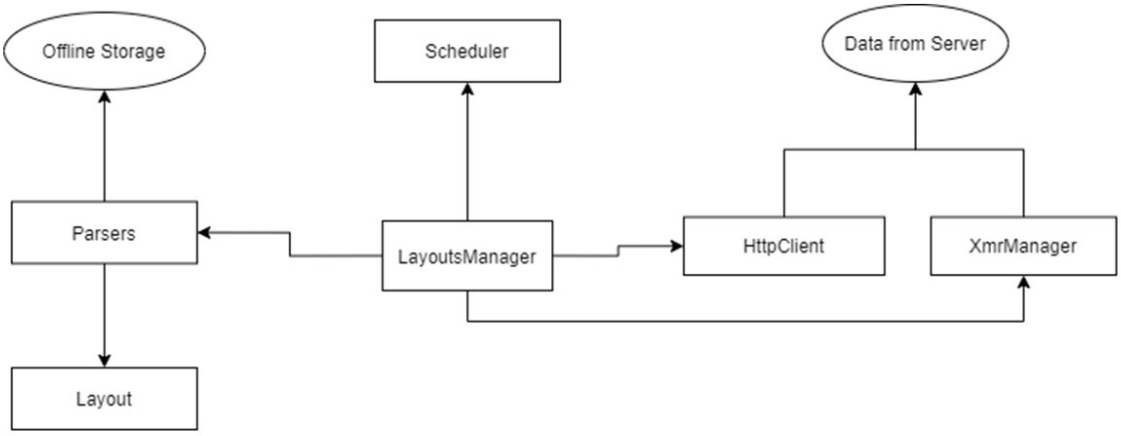
Xibo CMS



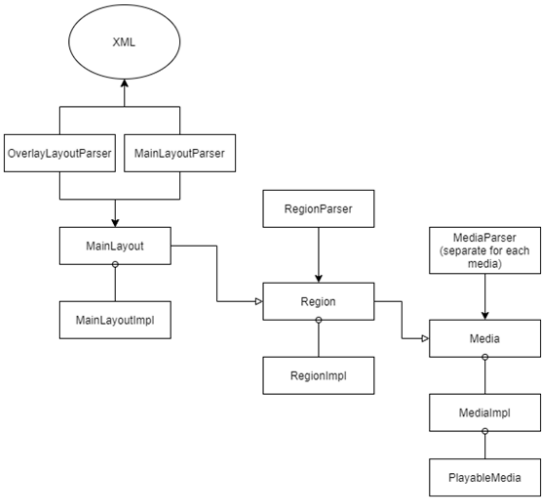
Інфраструктура плеєра



Загальна схема роботи плеєра



Обробка медіаконтента

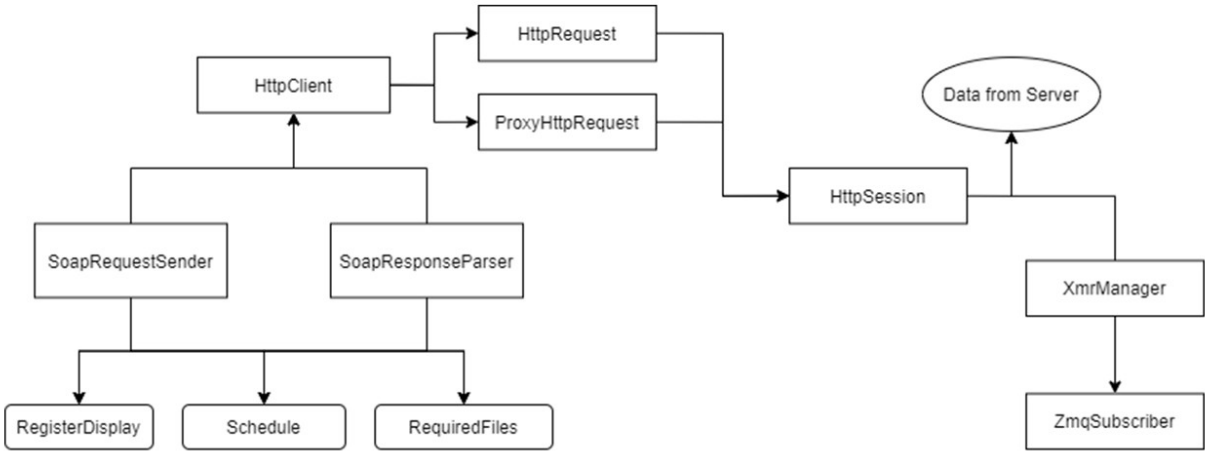


MainLayout – контейнер для регіонів, є центральним класом обробки контента

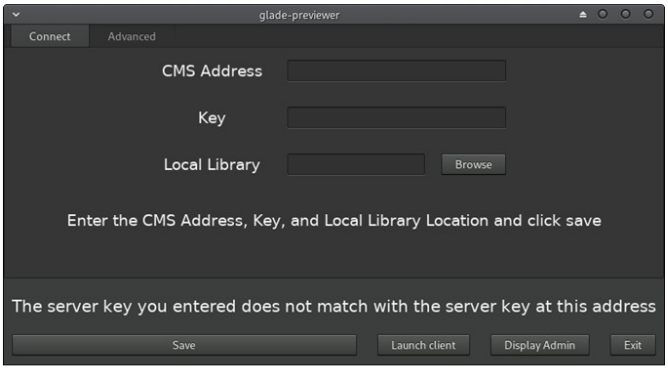
Region – контейнер для медіа, показує тільки одне медіа в конкретний момент часу

Media – представляє собою контент, який потрібно відтворяти на екрані (аудіо, відео, зображення або веб-контент)

Серверна взаємодія



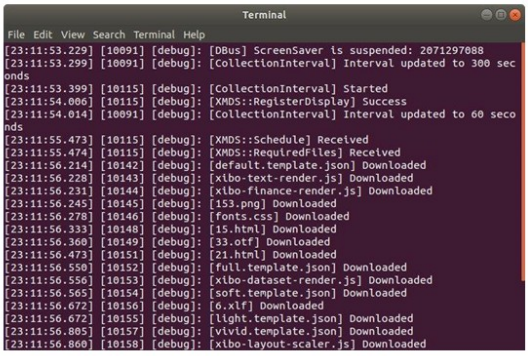
Конфігурація плеєра

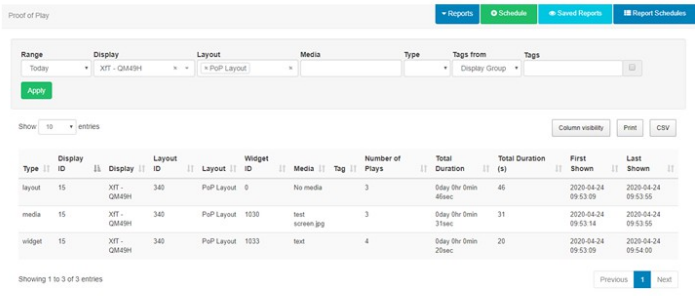


Додаток для конфігурації

Файли конфігурації

Ведення логів і статистика





Рівні ведення логів:
- Помилки
- Інформація
- Налагоджувальна інформація
- Додаткова інформація

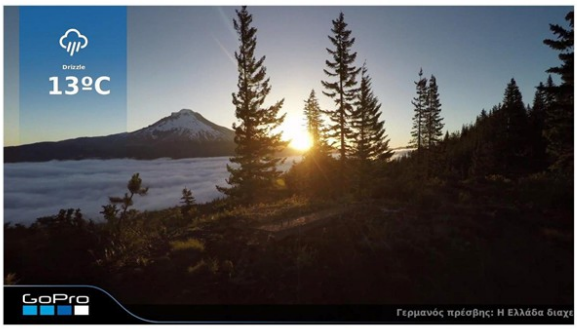
Типи статистичних даних:
- Макет
- Медіа

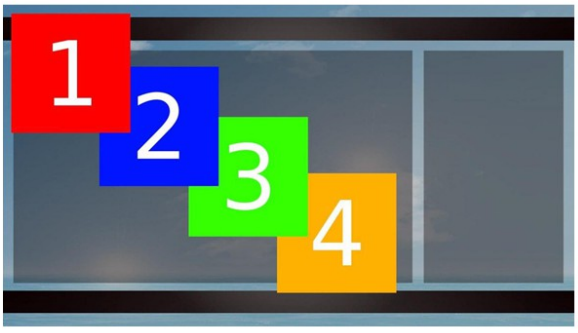
Player Status Screen

Збираємо всю необхідну інформацію, яка показує нам поточний стан плеєра.
Інформація розбита на три групи:

- Загальна інформація
- CMS інформація
- Інформація про планувальник
- XMR інформація

Види контента





- Відео
- Аудіо
- Зображення
- Веб-контент

Фотографія макету

Висновки

- отримані навички роботи в системі з обмеженими ресурсами;
- отримані навички роботи з бібліотеками для обробки медіаконтенту;
- отримані навички розробки інфраструктури в рамках Digital Signage;
- розроблена мережева архітектура для взаємодії з сервером;
- розроблена архітектура обробки медіаконтенту

Додаток Б

ЛІСТИНГИ ПРОГРАМНИХ КОМПОНЕНТІВ

Файл main.cpp

```
#include "MyApp.hpp"
#include "common/logger/Logging.hpp"

int main(int /*argc*/, char** /*argv*/)
{
    try
    {
        auto&& app = MyApp::create("org.gtkmm.project");
        return app.run();
    }
    catch (std::exception& e)
    {
        Log::error("[Main] {}", e.what());
    }

    return 0;
```

Реалізація MainLayout

```
#include "MainLayoutImpl.hpp"

#include "control/widgets/Image.hpp"
#include "control/widgets/OverlayContainer.hpp"

#include "common/logger/Logging.hpp"

namespace ph = std::placeholders;

MainLayoutImpl::MainLayoutImpl(const MainLayoutOptions& options)
:
    options_(options),
             view_(OverlayContainerFactory::create(options.width,
options.height))
{
                                                              view_-
>shown().connect(std::bind(&MainLayoutImpl::startLayout, this));
}

void
MainLayoutImpl::setBackground(std::shared_ptr<Xibo::Image>&&
background)
{
    if (background)
    {
        view_->setMainChild(background);
    }
}
```

```
void    MainLayoutImpl::addRegion(std::unique_ptr<Xibo::Region>&&
region, int left, int top, int zorder)
{
    assert(region);
    assert(region->view());

    view_->add(region->view(), left, top, zorder);
                                                              region-
>expired().connect(std::bind(&MainLayoutImpl::onRegionExpired,
this, ph::_1));
    monitorMediaStats(*region);

    regions_.emplace_back(std::move(region));
}

void MainLayoutImpl::monitorMediaStats(Xibo::Region& region)
{
    for (auto&& media : region.mediaList())
    {
        media->statEnabled(options_.statEnabled);
        if (media->statEnabled())
        {
            media->statReady().connect(
                    [id = media->id(), this](const PlayingStat&
interval) { mediaIntervals_.emplace(id, interval); });
        }
    }
}

SignalLayoutExpired& MainLayoutImpl::expired()
{
    return expired_;
}

SignalLayoutStatReady& MainLayoutImpl::statReady()
{
    return statsReady_;
}

SignalLayoutMediaStatsReady& MainLayoutImpl::mediaStatsReady()
{
    return mediaStatsReady_;
}

void MainLayoutImpl::restart()
{
    stopLayout();

    interval_.clear();
    mediaIntervals_.clear();
    expiredRegions_.clear();

    startLayout();
```

```
}

std::shared_ptr<Xibo::Widget> MainLayoutImpl::view()
{
    return view_;
}

int MainLayoutImpl::id() const
{
    return options_.id;
}

void MainLayoutImpl::onRegionExpired(int regionId)
{
    Log::trace("[MainLayout] Region {} expired", regionId);
    expiredRegions_.insert(regionId);

    if (areAllRegionsExpired())
    {
        onAllRegionsExpired();
    }
}

void MainLayoutImpl::onAllRegionsExpired()
{
    stopLayout();
    if (options_.statEnabled)
    {
        statsReady_(interval_);
    }
    if (!mediaIntervals_.empty())
    {
        mediaStatsReady_(mediaIntervals_);
    }

    expired_();
}

void MainLayoutImpl::startLayout()
{
    interval_.started = DateTime::now();
    startRegions();
}

void MainLayoutImpl::stopLayout()
{
    interval_.finished = DateTime::now();
    stopRegions();
}

bool MainLayoutImpl::areAllRegionsExpired() const
{
    return expiredRegions_.size() == regions_.size();
```

```
}

void MainLayoutImpl::startRegions()
{
    for (auto&& region : regions_)
    {
        region->start();
    }
}

void MainLayoutImpl::stopRegions()
{
    for (auto&& region : regions_)
    {
        region->stop();
    }
}
```

Реалізація Media

```
#include "control/media/MediaImpl.hpp"
#include "common/constants.hpp"

MediaImpl::MediaImpl(const MediaOptions& options) :
    options_(options),
    timer_(std::make_unique<Timer>()),
    statEnabled_(false),
    playing_(false)
{
    assert(timer_);

                              if       (options_.statPolicy       !=
MediaOptions::StatPolicy::Inherit)
    {
                         statEnabled_   =   options_.statPolicy   ==
MediaOptions::StatPolicy::Enable ? true : false;
    }
}

void   MediaImpl::setWidget(const   std::shared_ptr<Xibo::Widget>&
widget)
{
    widget_ = widget;
}

void MediaImpl::attach(std::unique_ptr<Media>&& attachedMedia)
{
    attachedMedia_ = std::move(attachedMedia);
}

bool MediaImpl::playing() const
{
    return playing_;
}
```

```
void MediaImpl::start()
{
    if (playing_) return;

    stat_.clear();

    playing_ = true;
    stat_.started = DateTime::now();
    startTimer(options_.duration);
    startAttachedMedia();
    onStarted();
}

void MediaImpl::startTimer(int duration)
{
    if (duration > 0)
    {
         timer_->startOnce(std::chrono::seconds(duration), [this]
{ finished_(); });
    }
}

void MediaImpl::startAttachedMedia()
{
    if (attachedMedia_)
    {
        attachedMedia_->start();
    }
}

void MediaImpl::onStarted()
{
    if (widget_)
    {
        widget_->show();
    }
}

void MediaImpl::stop()
{
    if (!playing_) return;

    playing_ = false;
    stat_.finished = DateTime::now();
    timer_->stop();
    stopAttachedMedia();
    onStopped();
    statReady_(stat_);
}

void MediaImpl::statEnabled(bool enable)
{
```

```
                            if       (options_.statPolicy       ==
MediaOptions::StatPolicy::Inherit)
    {
        statEnabled_ = enable;
    }
}

bool MediaImpl::statEnabled() const
{
    return statEnabled_;
}

MediaOptions::StatPolicy MediaImpl::statPolicy() const
{
    return options_.statPolicy;
}

int MediaImpl::id() const
{
    return options_.id;
}

void
MediaImpl::inTransition(std::unique_ptr<TransitionExecutor>&&
transition)
{
    inTransition_ = std::move(transition);
}

void
MediaImpl::outTransition(std::unique_ptr<TransitionExecutor>&&
transition)
{
    outTransition_ = std::move(transition);
}

void MediaImpl::stopAttachedMedia()
{
    if (attachedMedia_)
    {
        attachedMedia_->stop();
    }
}

void MediaImpl::applyInTransition()
{
    if (inTransition_)
    {
        inTransition_->apply();
    }
}

void MediaImpl::onStopped()
```

```
{
    if (widget_)
    {
        widget_->hide();
    }
}

SignalMediaFinished& MediaImpl::finished()
{
    return finished_;
}

SignalMediaStatReady& MediaImpl::statReady()
{
    return statReady_;
}

MediaGeometry::Align MediaImpl::align() const
{
    return options_.geometry.align;
}

MediaGeometry::Valign MediaImpl::valign() const
{
    return options_.geometry.valign;
}

std::shared_ptr<Xibo::Widget> MediaImpl::view()
{
    return widget_;
}
```

Реалізація XmrManager

```
#include "XmrManager.hpp"
#include "MainLoop.hpp"
#include "common/parsing/Parsing.hpp"
#include "common/crypto/RsaManager.hpp"
#include "common/dt/DateTime.hpp"
#include "common/logger/Logging.hpp"
#include "config/AppConfig.hpp"

const size_t CHANNEL_PART = 0;
const size_t KEY_PART = 1;
const size_t MESSAGE_PART = 2;
const char* const HearbeatChannel = "H";

XmrManager::XmrManager(const XmrChannel& mainChannel) :
mainChannel_(static_cast<std::string>(mainChannel)) {}

// TODO: strong type
void XmrManager::connect(const std::string& host)
{
    if (info_.host == host) return;
```

```
    info_.host = host;
    subscriber_.messageReceived().connect(
        [this](const Zmq::MultiPartMessage& message)
{ processMultipartMessage(message); });
    subscriber_.run(host, Zmq::Channels{mainChannel_,
HearbeatChannel});
}

void XmrManager::stop()
{
    subscriber_.stop();
}

CollectionIntervalAction& XmrManager::collectionInterval()
{
    return collectionIntervalAction_;
}

ScreenshotAction& XmrManager::screenshot()
{
    return screenshotAction_;
}

XmrStatus XmrManager::status()
{
    return info_;
}

void XmrManager::processMultipartMessage(const
Zmq::MultiPartMessage& multipart)
{
    if (multipart[CHANNEL_PART] == mainChannel_)
    {
        try
        {
            auto decryptedMessage =
decryptMessage(multipart[KEY_PART], multipart[MESSAGE_PART]);
            auto xmrMessage = parseMessage(decryptedMessage);

            processXmrMessage(xmrMessage);

            info_.lastMessageDt = DateTime::now();
        }
        catch (CryptoPP::Exception& e)
        {
            Log::error("[XMR::Crypto] {}. You need to
reconfigure XMR for this display in the CMS and wait for the "
                       "next collection "
                       "interval so all keys will be updated.",
                       e.what());
        }
        catch (std::exception& e)
```

```
        {
            Log::error("[XMR] {}", e.what());
        }
    }
    else
    {
        info_.lastHeartbeatDt = DateTime::now();
    }
}

std::string XmrManager::decryptMessage(const std::string&
encryptedBase64Key, const std::string& encryptedBase64Message)
{
    auto privateKey = RsaManager::instance().privateKey();

    auto encryptedKey =
CryptoUtils::fromBase64(encryptedBase64Key);
    auto messageKey =
CryptoUtils::decryptPrivateKeyPkcs(encryptedKey, privateKey);

    auto encryptedMessage =
CryptoUtils::fromBase64(encryptedBase64Message);

    return CryptoUtils::decryptRc4(encryptedMessage,
messageKey);
}

XmrMessage XmrManager::parseMessage(const std::string&
jsonMessage)
{
    auto tree = Parsing::jsonFromString(jsonMessage);

    XmrMessage message;
    message.action = tree.get<std::string>("action");
    message.createdDt =
DateTime::fromIsoExtendedString(tree.get<std::string>("createdDt
"));
    message.ttl = tree.get<int>("ttl");

    return message;
}

void XmrManager::processXmrMessage(const XmrMessage& message)
{
    if (isMessageExpired(message)) return;

    if (message.action == "collectNow")
    {
        MainLoop::pushToUiThread([this]()
{ collectionIntervalAction_(); });
    }
    else if (message.action == "screenShot")
    {
```

```
        MainLoop::pushToUiThread([this]() { screenshotAction_();
});
    }
}

bool XmrManager::isMessageExpired(const XmrMessage& message)
{
    auto resultDt = message.createdDt +
DateTime::Seconds(message.ttl);
    if (resultDt < DateTime::nowUtc())
    {
        return true;
    }
    return false;
}
```

Реалізація CollectionInterval

```
#include "CollectionInterval.hpp"
#include "MainLoop.hpp"
#include "NotifyStatusInfo.hpp"
#include "common/PlayerRuntimeError.hpp"
#include "common/dt/DateTime.hpp"
#include "common/dt/Timer.hpp"
#include "common/fs/FileSystem.hpp"
#include "common/fs/StorageUsageInfo.hpp"
#include "common/logger/Logging.hpp"
#include "common/logger/XmlLogsRetriever.hpp"
#include "common/storage/FileCache.hpp"
#include "common/system/System.hpp"
#include "config/AppConfig.hpp"

#include "cms/xmds/XmdsRequestSender.hpp"
#include "stat/Recorder.hpp"
#include "stat/records/XmlFormatter.hpp"

namespace ph = std::placeholders;

CollectionInterval::CollectionInterval(XmdsRequestSender&
xmdsSender,
                                       Stats::Recorder&
statsRecorder,
                                       FileCache& fileCache,
                                       const FilePath&
resourceDirectory) :
    xmdsSender_{xmdsSender},
    statsRecorder_{statsRecorder},
    fileCache_{fileCache},
    intervalTimer_{std::make_unique<Timer>()},
    collectInterval_{DefaultInterval},
    running_{false},
    status_{},
    currentLayoutId_{EmptyLayoutId},
    resourceDirectory_{resourceDirectory}
```

```
{
    assert(intervalTimer_);
}

bool CollectionInterval::running() const
{
    return running_;
}

void CollectionInterval::stop()
{
    workerThread_.reset();
}

void CollectionInterval::startTimer()
{
    intervalTimer_-
>startOnce(std::chrono::seconds(collectInterval_), [this]()
{ collectNow(); });
}

void CollectionInterval::collectNow()
{
    if (!running_)
    {
        running_ = true;
        workerThread_ = std::make_unique<JoinableThread>([=]() {
            Log::debug("[CollectionInterval] Started");

            auto registerDisplayResult =

xmdsSender_.registerDisplay(AppConfig::codeVersion(),
AppConfig::releaseVersion(), "Display").get();
            onDisplayRegistered(registerDisplayResult);
        });
    }
}

void CollectionInterval::sessionFinished(const PlayerError&
error)
{
    running_ = false;
    startTimer();
    Log::debug("[CollectionInterval] Finished. Next collection
will start in {} seconds", collectInterval_);

    MainLoop::pushToUiThread([this, error]()
{ collectionFinished_(error); });
}

void CollectionInterval::onDisplayRegistered(const
ResponseResult<RegisterDisplay::Result>& registerDisplay)
{
```

```
    auto [error, result] = registerDisplay;
    if (!error)
    {
        auto displayError = displayStatus(result.status);
        if (!displayError)
        {
            Log::debug("[XMDS::RegisterDisplay] Success");

            status_.registered = true;
            status_.lastChecked = DateTime::now();

            MainLoop::pushToUiThread([this, result =
std::move(result.playerSettings)]()
{ settingsUpdated_(result); });

            auto requiredFilesResult =
xmdsSender_.requiredFiles().get();
            auto scheduleResult = xmdsSender_.schedule().get();

            onSchedule(scheduleResult);
            onRequiredFiles(requiredFilesResult);

            submitLogs();
            submitStats();
            notifyStatus();
        }
        sessionFinished(displayError);
    }
    else
    {
        sessionFinished(error);
    }
}

void CollectionInterval::setCurrentLayoutId(const LayoutId&
currentLayoutId)
{
    currentLayoutId_ = currentLayoutId;
}

PlayerError CollectionInterval::displayStatus(const
RegisterDisplay::Result::Status& status)
{
    using DisplayCode = RegisterDisplay::Result::Status::Code;

    switch (status.code)
    {
        case DisplayCode::Ready: return {};
        case DisplayCode::Added:
        case DisplayCode::Waiting: return {"CMS",
status.message};
        default: return {"CMS", "Unknown error with
RegisterDisplay"};
```

```
    }
}

void CollectionInterval::updateInterval(int collectInterval)
{
    if (collectInterval_ != collectInterval)
    {
        Log::debug("[CollectionInterval] Interval updated to {}
seconds", collectInterval);
        collectInterval_ = collectInterval;
    }
}

// TODO potential data race here
CmsStatus CollectionInterval::status() const
{
    return status_;
}

SignalSettingsUpdated& CollectionInterval::settingsUpdated()
{
    return settingsUpdated_;
}

SignalScheduleAvailable& CollectionInterval::scheduleAvailable()
{
    return scheduleAvailable_;
}

SignalCollectionFinished&
CollectionInterval::collectionFinished()
{
    return collectionFinished_;
}

SignalFilesDownloaded& CollectionInterval::filesDownloaded()
{
    return filesDownloaded_;
}

void CollectionInterval::onRequiredFiles(const
ResponseResult<RequiredFiles::Result>& requiredFiles)
{
    auto [error, result] = requiredFiles;
    if (!error)
    {
        Log::debug("[XMDS::RequiredFiles] Received");

        RequiredFilesDownloader downloader{xmdsSender_,
fileCache_};

        auto&& files = result.requiredFiles();
        auto&& resources = result.requiredResources();
```

```
        status_.requiredFiles = files.size() + resources.size();

        auto resourcesResult = downloader.download(resources);
        auto filesResult = downloader.download(files);

        resourcesResult.wait();
        filesResult.wait();

        updateMediaInventory(result);

        MainLoop::pushToUiThread([this]()
{ filesDownloaded_(); });
    }
    else
    {
        sessionFinished(error);
    }
}

void CollectionInterval::updateMediaInventory(const
RequiredFiles::Result& result)
{
    MediaInventoryItems items;
    for (auto&& file : result.requiredFiles())
    {
        items.emplace_back(file, fileCache_.valid(file.name()));
    }
    for (auto&& file : result.requiredResources())
    {
        items.emplace_back(file, fileCache_.valid(file.name()));
    }
    onSubmitted("MediaInventory",
xmdsSender_.mediaInventory(std::move(items)).get());
}

void CollectionInterval::onSchedule(const
ResponseResult<Schedule::Result>& schedule)
{
    auto [error, result] = schedule;
    if (!error)
    {
        Log::debug("[XMDS::Schedule] Received");
        MainLoop::pushToUiThread([this, result =
std::move(result)]() {

scheduleAvailable_(LayoutSchedule::fromString(result.scheduleXml
));
        });
    }
    else
    {
        sessionFinished(error);
```

```
    }
}

void CollectionInterval::submitLogs()
{
    XmlLogsRetriever logsRetriever;
    auto submitLogsResult =
xmdsSender_.submitLogs(logsRetriever.retrieveLogs()).get();
    onSubmitted("SubmitLogs", submitLogsResult);
}

void CollectionInterval::submitStats()
{
    try
    {
        const auto recordsCount = statsRecorder_.recordsCount();
        if (recordsCount > 0)
        {
            const auto RecordsToSend = [recordsCount]() ->
size_t {
                if (recordsCount > 500)
                    return 300;
                else
                    return recordsCount > 50 ? 50 :
recordsCount;
            }();

            Log::debug("[CollectionInterval] Total records: {}
Records to send {}", recordsCount, RecordsToSend);

            auto records =
statsRecorder_.records(RecordsToSend);
            statsRecorder_.removeFromQueue(RecordsToSend);

            Stats::XmlFormatter formatter;
            auto submitStatsResult =
xmdsSender_.submitStats(formatter.format(records)).get();
            onSubmitted("SubmitStats", submitStatsResult);
        }
    }
    catch (const std::exception& e)
    {
        Log::error(e.what());
        Log::error("[CollectionInterval] Failed to submit
stats");
    }
}

void CollectionInterval::notifyStatus()
{
    NotifyStatusInfo notifyInfo;
    // FIXME: store it in collection interval until XMDS
refactoring
```

```
    notifyInfo.currentLayoutId = currentLayoutId_;
    notifyInfo.deviceName = System::hostname();
    notifyInfo.spaceUsageInfo =
FileSystem::storageUsageFor(resourceDirectory_);
    notifyInfo.timezone = DateTime::currentTimezone();

    auto notifyStatusResult =
xmdsSender_.notifyStatus(notifyInfo.string()).get();
    onSubmitted("NotifyStatus", notifyStatusResult);
}

template <typename Result>
void CollectionInterval::onSubmitted(std::string_view
requestName, const ResponseResult<Result>& submitResult)
{
    auto [error, result] = submitResult;
    if (!error)
    {
        if (result.success)
        {
            Log::debug("[XMDS::{}] Submitted", requestName);
        }
        else
        {
            Log::error("[XMDS::{}] Not submited due to unknown
error", requestName);
        }
    }
    else
    {
        sessionFinished(error);
    }
}
```