```
#include "add_new_descriptor.hpp"
#include "ui_add_new_descriptor.h"
#include <QFileDialog>
#include "descriptor.hpp"
#include <QFile>
#include <QJsonDocument>
#include <QJsonObject>
#include <QJsonArray>
#include <QDebug>
#include <QMessageBox>
#include <QFile>
#include <QIODevice>


Add_New_Descriptor::Add_New_Descriptor(QString Librarypath, QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::Add_New_Descriptor)
{
    ui->setupUi(this);
    ui->Image_path_hidden->setVisible(false);
    qDebug() << "Add_New_Descriptor constructor called";
    qDebug() << "Library path: " << Librarypath;

    setLibraryPath(Librarypath);

    // Ensure no duplicate connections
    disconnect(ui->loadImageButton, nullptr, nullptr, nullptr);
    disconnect(ui->save_the_descriptor, nullptr, nullptr, nullptr);

    // Connect signals to slots
    connect(ui->loadImageButton, &QPushButton::clicked, this, &Add_New_Descriptor::on_loadI
mageButton_clicked);
    connect(ui->save_the_descriptor, &QPushButton::clicked, this, &Add_New_Descriptor::on_s
ave_the_descriptor_clicked);

    // Apply styles to the UI elements
    ui->loadImageButton->setStyleSheet("background-color: #99c1f1; color: white; padding: 1
0px 20px; border: none; border-radius: 5px;");
    ui->save_the_descriptor->setStyleSheet("background-color: #99c1f1; color: white; paddin
g: 10px 20px; border: none; border-radius: 5px;");
    ui->add_title->setStyleSheet("padding: 10px; border: 1px solid #ccc; border-radius: 5px
;");
    ui->add_source->setStyleSheet("padding: 10px; border: 1px solid #ccc; border-radius: 5p
x;");

    ui->add_cost->setStyleSheet("padding: 10px; border: 1px solid #ccc; border-radius: 5px;
");
    ui->add_cost->setPlaceholderText("Enter the cost");
    ui->add_title->setPlaceholderText("Enter the title");
    ui->add_source->setPlaceholderText("Enter the source");

    // ui->ImageLabel->setStyleSheet("border: 1px solid #ccc; padding: 5px; background-imag
e: url(Images/galerie.jpg); background-repeat: no-repeat; background-position: center;");

    ui->ImageLabel->show();
    ui->comboBox_Access->addItem("L");
    ui->comboBox_Access->addItem("O");
}

Add_New_Descriptor::~Add_New_Descriptor()
{
    delete ui;
}

void Add_New_Descriptor::on_loadImageButton_clicked()
{
    static bool isProcessing = false;
    if (isProcessing) {
```

```
        qDebug() << "on_loadImageButton_clicked: Already processing!";
        return; // Prevent re-entry if already processing
    }
    isProcessing = true;

    qDebug() << "on_loadImageButton_clicked: Dialog opened";
    QString imagePath = QFileDialog::getOpenFileName(this, "Open Image", "", "Image Files (
*.png *.jpg *.bmp)");

    if (!imagePath.isEmpty()) {
        QPixmap pixmap(imagePath);
        ui->Image_path_hidden->setText(imagePath);
        qDebug() << "Selected imagePath: " << imagePath;
        ui->ImageLabel->setPixmap(pixmap.scaled(210, 210, Qt::KeepAspectRatio));
    } else {
        qDebug() << "on_loadImageButton_clicked: No file selected";
    }

    isProcessing = false; // Reset the flag
}

void Add_New_Descriptor::on_save_the_descriptor_clicked()
{
    static bool isProcessing = false;
    if (isProcessing) {
        qDebug() << "on_save_the_descriptor_clicked: Already processing!";
        return; // Prevent re-entry if already processing
    }
    isProcessing = true;

    // Gather data from UI
    QString title = ui->add_title->text();
    QString source = ui->add_source->text();
    QString cost = ui->add_cost->text();
    QString imagePath = ui->Image_path_hidden->text();
    char access = ui->comboBox_Access->currentText().toStdString().c_str()[0];

    qDebug() << "on_save_the_descriptor_clicked: Gathering data from UI";
    qDebug() << "Access: " << access;
    qDebug() << "Image path: " << imagePath;

    // Verify if the image path is valid
    if (imagePath.isEmpty()) {
        QMessageBox::warning(this, "Input Error", "Please load an image before saving.");
        isProcessing = false;
        return;
    }

    QString appPath = QCoreApplication::applicationDirPath();
    QString destinationDir = appPath + "/Images/";
    // QString destinationPath = destinationDir + QFileInfo(imagePath).fileName();

    // Ensure the destination directory exists
    QDir dir(destinationDir);
    if (!dir.exists()) {
        if (!dir.mkpath(".")) {
            QMessageBox::warning(this, "Directory Error", "Could not create destination dir
ectory.");
            isProcessing = false;
            return;
        }
    }

    QString fileName = QFileInfo(imagePath).fileName();
    QString uniqueFileName = fileName;
    int counter = 2;

    while (QFile::exists(destinationDir + uniqueFileName)) {
```

```
        uniqueFileName = QFileInfo(fileName).completeBaseName() + "_" + QString::number(cou
nter) + "." + QFileInfo(fileName).suffix();
        counter++;
    }

    QString destinationPath = destinationDir + uniqueFileName;
    // Copy the image to the destination directory
    QFile file(imagePath);
    if (!file.copy(destinationPath)) {
        QMessageBox::warning(this, "File Error", "Could not copy the image file.");
        isProcessing = false;
        return;
    }

    // Image image("/Images/" + QFileInfo(imagePath).fileName());
    Image image("/Images/" + uniqueFileName);

    Descriptor descriptor(0, cost.toDouble(), title, source, access, image);

    // Save descriptor to JSON
    QFile jsonFile(Librarypath);
    if (!jsonFile.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, "File Error", "Could not open the library file.");
        isProcessing = false;
        return;
    }

    QJsonDocument doc = QJsonDocument::fromJson(jsonFile.readAll());
    jsonFile.close();

    QJsonObject obj = doc.object();
    QJsonArray array = obj["library"].toArray();

    QJsonObject newDescriptor;
    newDescriptor["id"] = array.size() + 1;
    newDescriptor["cost"] = cost.toDouble();
    newDescriptor["title"] = title;
    newDescriptor["source"] = source;
    newDescriptor["access"] = QString(access);
    // newDescriptor["Imagepath"] = "/Images/" + QFileInfo(imagePath).fileName();
    newDescriptor["Imagepath"] = "/Images/" + uniqueFileName;


    array.append(newDescriptor);
    obj["library"] = array;

    if (!jsonFile.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(this, "File Error", "Could not open the library file for writi
ng.");
        isProcessing = false;
        return;
    }

    jsonFile.write(QJsonDocument(obj).toJson());
    jsonFile.close();

    // Close the dialog
    accept();
    qDebug() << "on_save_the_descriptor_clicked: Descriptor saved successfully";

    isProcessing = false; // Reset the flag
}

void Add_New_Descriptor::setLibraryPath(QString Librarypath)
{
    this->Librarypath = Librarypath;
}
```

```
#include "descriptor.hpp"
#include <QJsonArray>

Descriptor::Descriptor(const Image& img)
    : idDes(0), cost(0), title("UKNOWN"),
      source("UKNOWN"), access('L'), image(img), nextDescriptor(nullptr) {}

Descriptor::Descriptor(int idDesc, const Image& img)
    : idDes(idDesc), cost(0), title("UKNOWN"),
      source("UKNOWN"), access('L'), image(img), nextDescriptor(nullptr) {}

Descriptor::Descriptor(int idDesc, double costValue, const Image& img)
    : idDes(idDesc), cost(costValue), title("UKNOWN"),
      source("UKNOWN"), access('L'), image(img), nextDescriptor(nullptr) {}

Descriptor::Descriptor(int idDesc, double costValue, const QString& descTitle, const Image&
 img)
    : idDes(idDesc), cost(costValue), title(descTitle),
      source("UKNOWN"), access('L'), image(img), nextDescriptor(nullptr) {}

Descriptor::Descriptor(int idDesc, double costValue, const QString& descTitle,
                       const QString& descSource, const Image& img)
    : idDes(idDesc), cost(costValue), title(descTitle),
      source(descSource), access('L'), image(img), nextDescriptor(nullptr) {}

Descriptor::Descriptor(int idDesc, double costValue, const QString& descTitle,
                       const QString& descSource, const char descAccess, const Image& img)
    : idDes(idDesc), cost(costValue), title(descTitle),
      source(descSource), access(descAccess), image(img), nextDescriptor(nullptr) {}




unsigned int Descriptor::getIdDescriptor() const { return this->idDes; }
double Descriptor::getCost() const { return this->cost; }
QString Descriptor::getTitle() const { return this->title; }
QString Descriptor::getSource() const { return this->source; }
char Descriptor::getAccess() const { return this->access; }
Image Descriptor::getImage() const { return this->image; }

Descriptor* Descriptor::getNextDescriptor() const { return this->nextDescriptor; }

void Descriptor::setIdDescriptor(int newIdDes) { this->idDes = newIdDes; }
void Descriptor::setCost(double newCost)  { this->cost = newCost; }
void Descriptor::setTitle(const QString& descTitle) { this->title = descTitle; }
void Descriptor::setSource(const QString& descSource) { this->source = descSource; }
void Descriptor::setAccess(const char& descAccess) { this->access = descAccess; }
void Descriptor::setImage(const Image& img) { this->image = img; }

void Descriptor::setNextDescriptor(Descriptor* nextDesc) { this->nextDescriptor = nextDesc;
 }

QPixmap Descriptor::cvMatToQPixmap(const cv::Mat &mat) const {
    // Step 1: Convert cv::Mat to QImage
    QImage img;
    if (mat.channels() == 1) {
        // Grayscale image
        img = QImage(mat.data, mat.cols, mat.rows, mat.step, QImage::Format_Grayscale8);
    } else if (mat.channels() == 3) {
        // Convert BGR to RGB
        cv::Mat rgb;
        cv::cvtColor(mat, rgb, cv::COLOR_BGR2RGB);
        img = QImage(rgb.data, rgb.cols, rgb.rows, rgb.step, QImage::Format_RGB888);
    } else if (mat.channels() == 4) {
        // Direct BGRA to QImage with alpha
        img = QImage(mat.data, mat.cols, mat.rows, mat.step, QImage::Format_ARGB32);
    } else {
        throw std::invalid_argument("Unsupported cv::Mat format.");
```

```
    }

    // Ensure the QImage is valid before converting to QPixmap
    if (img.isNull()) {
        throw std::runtime_error("Failed to convert cv::Mat to QImage");
    }

    // Step 2: Convert QImage to QPixmap
    QPixmap pixmap = QPixmap::fromImage(img);
    if (pixmap.isNull()) {
        throw std::runtime_error("Failed to convert QImage to QPixmap");
    }

    return pixmap;
}
cv::Mat Descriptor::QPixmapToCvMat(const QPixmap &pixmap) const {
    QImage qImage = pixmap.toImage().convertToFormat(QImage::Format_ARGB32);

    cv::Mat mat(qImage.height(), qImage.width(), CV_8UC4, const_cast<uchar*>(qImage.bits())
, qImage.bytesPerLine());

    // Convert ARGB to BGR
    cv::Mat matBGR;
    cv::cvtColor(mat, matBGR, cv::COLOR_BGRA2BGR);

    return matBGR;
}
void Descriptor::display() const {
    qDebug()  << "Descriptor ID: " << idDes ;

}


QJsonObject Descriptor::toJson() const {
    QJsonObject json;
    json["Imagepath"] = image.getPath();
    json["access"] = QString(access);
    json["cost"] = cost;
    json["id"] = static_cast<int>(idDes);
    json["source"] = source;
    json["title"] = title;

    return json;
}
```

```
#include "descriptordetails.hpp"
#include "ui_descriptordetails.h"
#include <QMessageBox>
#include "imageproccessing.hpp"
#include "ClickableLabel.hpp"
#include <QFile>
#include <QJsonArray>
#include <QJsonDocument>
#include <QFileDialog>

DescriptorDetails::DescriptorDetails(QWidget *parent , bool access,QString LibraryPath )

    : QDialog(parent)
    , ui(new Ui::DescriptorDetails)
    , currentDescriptor(nullptr)
    , access(access)
{
    ui->setupUi(this);

        ui->comboBox->addItem("Gaussien Filter");
        ui->comboBox->addItem("Median Filter");
        ui->comboBox->addItem("To GrayScale");
        ui->comboBox->addItem("Edge Detection");
        ui->comboBox->addItem("Seuillage");
        ui->comboBox->addItem("Rotation");
        ui->comboBox_2->addItem("Left");
        ui->comboBox_2->addItem("Right");
        ui->comboBox_2->addItem("Down");
        ui->comboBox_2->addItem("Up");
        ui->comboBox->addItem("SIFT");
        ui->comboBox->addItem("Histogram");
        ui->comboBox->addItem("Erosion");


        ui->filtreButton->setVisible(true);
        ui->comboBox->setVisible(true);
        ui->FilteredImageLabel->setVisible(true);

        ui->thresholdLabel->setText("Threshold:");
        ui->thresholdLabel->setVisible(false);
        ui->thresholdInput->setVisible(false);
        ui->comboBox_2->setVisible(false);

        ui->kernelsizelabel->setText("Kernel size");
        ui->kernelsizelabel->setVisible(false);
        ui->Kernelsizeinput->setVisible(false);

        connect(ui->comboBox, QOverload<int>::of(&QComboBox::currentIndexChanged), this, &D
escriptorDetails::onFilterSelectionChanged);
        connect(ui->FilteredImageLabel, &ClickableLabel::clicked, this, [this]() {onLabelCl
icked(ui->FilteredImageLabel);});
        connect(ui->ImageLabel, &ClickableLabel::clicked, this, [this]() {onLabelClicked(ui
->ImageLabel);});

        QPixmap pixmap(":/AppImages/traiter.png");
        pixmap = pixmap.scaled(ui->label_icone_1->size(), Qt::KeepAspectRatio, Qt::SmoothTr
ansformation); // Redimensionne l'image tout en gardant les proportions
        ui->label_icone_1->setPixmap(pixmap);
        ui->label_icone_2->setPixmap(pixmap);

}


DescriptorDetails::~DescriptorDetails()
{
    delete ui;
}
```

```
void DescriptorDetails::setLibraryPath(QString libraryPath){
    this->LibraryPath = libraryPath;
}

QString DescriptorDetails::getLibraryPath(){
    return this->LibraryPath;
}

void DescriptorDetails::setDescriptor(Descriptor* descriptor) {
    currentDescriptor = descriptor; // Store the current descriptor

    // ui->idLabel->setText(QString::number(descriptor->getIdDescriptor()));
    // ui->costLabel->setText(QString::number(descriptor->getCost()));
    // ui->titleLabel->setText(descriptor->getTitle());
    // ui->sourceLabel->setText(descriptor->getSource());
    // ui->accessLabel->setText(QString(descriptor->getAccess()));
    QString appPath = QCoreApplication::applicationDirPath();

    // Load the image directly from the file path
    QPixmap pixmap(appPath + descriptor->getImage().getPath());
    if (pixmap.isNull()) {
        QMessageBox::warning(this, "Error", "Failed to load the image. Check the file path
or format.");
        return;
    }

    ui->ImageLabel->setPixmap(pixmap.scaled(ui->ImageLabel->size(), Qt::KeepAspectRatio, Qt
::SmoothTransformation));
    ui->ImageLabel->setAlignment(Qt::AlignCenter);
 if(access){
    // Clear the filtered image label
    ui->FilteredImageLabel->clear();
    ui->FilteredImageLabel->setPixmap(QPixmap());
 }
}

void DescriptorDetails::onFilterSelectionChanged(int index) {
    QString selectedFilter = ui->comboBox->itemText(index);

    // Si le filtre "Seuillage" est sÃ©lectionnÃ©, afficher le champ de saisie du seuil
    if (selectedFilter == "Seuillage") {
        ui->thresholdLabel->setVisible(true);
        ui->thresholdInput->setVisible(true);
    } else {
        // Cacher le champ pour tous les autres filtres
        ui->thresholdLabel->setVisible(false);
        ui->thresholdInput->setVisible(false);
    }
    if (selectedFilter == "Rotation") {
        ui->comboBox_2->setVisible(true);
    } else {
        // Cacher le champ pour tous les autres filtres
        ui->comboBox_2->setVisible(false);
    }
    if (selectedFilter =="Erosion") {
        ui->Kernelsizeinput->setVisible(true);
        ui->kernelsizelabel->setVisible(true);
    } else {
        // Cacher le champ pour tous les autres filtres
        ui->Kernelsizeinput->setVisible(false);
        ui->kernelsizelabel->setVisible(false);
    }

}


void DescriptorDetails::on_filtreButton_clicked() {
    QString filter = ui->comboBox->currentText();
```

```
    QString Rotate = ui->comboBox_2->currentText();

    ui->FilteredImageLabel->clear();

    QString imagePath = currentDescriptor->getImage().getPath();

    QString appPath = QCoreApplication::applicationDirPath();
    imagePath = appPath + imagePath;
    Mat inputImage = cv::imread(imagePath.toStdString());

    if (inputImage.empty()) {
        QMessageBox::warning(this, "Error", "Failed to convert QPixmap to cv::Mat.");
        return;
    }

    // Apply the selected filter
    ImageProccessing processor;
    try {
        Mat outputImage;

        if (filter == "Gaussien Filter") {
            // // Apply Gaussian filter with 5x5 kernel and sigma = 1.0
            outputImage = processor.applyGaussianFilter(inputImage);
            // imwrite(outputImage,"test.jpg")
        } else if (filter == "Edge Detection") {
            // Apply edge detection with 3*3 sobel filter
            outputImage = processor.applyEdgeDetection(inputImage);

        } else if (filter == "Median Filter") {
            // Apply Median filter with kernel size = 5
            outputImage = processor.applyCustomMedianFilter(inputImage, 3);

        } else if (filter == "Rotation") {

            if(Rotate=="Down"){
                outputImage = processor.rotateImage(inputImage, 180);
            }else if(Rotate=="Up"){
                outputImage = processor.rotateImage(inputImage, 0);
            }else if(Rotate=="Left"){
                outputImage = processor.rotateImage(inputImage, 270);
            }else if(Rotate=="Right"){
                outputImage = processor.rotateImage(inputImage, 90);
            }

        } else if (filter == "To GrayScale") {
            // Convert to grayscale
            outputImage = processor.toGrayScale(inputImage);

        } else if (filter == "SIFT") {
            outputImage = processor.applySIFT(inputImage);

        } else if (filter == "Seuillage") {

            int thresholdValue ;
            if (ui->thresholdInput->isVisible()) {
                bool ok;
                thresholdValue = ui->thresholdInput->text().toInt(&ok);
                if (!ok) {
                    QMessageBox::warning(this, "Erreur", "Valeur de seuil invalide.");
                    return;
                }
            }
            outputImage = processor.applyThreshold(inputImage,thresholdValue);

        } else if (filter == "Histogram") {
            // Calcul de l'histogramme
            outputImage = processor.calculateHistogram(inputImage);
```

```
        } else if (filter =="Erosion") {

            int Kernelsize ;
            if (ui->Kernelsizeinput->isVisible()) {
                bool ok;
                Kernelsize = ui->Kernelsizeinput->text().toInt(&ok);
                if (!ok) {
                    QMessageBox::warning(this, "Erreur", "Valeur de seuil invalide.");
                    return;
                }
            }
            outputImage = processor.applyErosion(inputImage,Kernelsize);

        } else if (filter =="Erosion"){
            outputImage = processor.applyErosion(inputImage,25);

        } else {
            throw invalid_argument("Invalid filter selected.");
        }

        QImage filteredQImage;

        if (outputImage.channels() == 4) { // ARGB image
            cv::Mat clonedImage = outputImage.clone(); // Clone ensures memory remains vali
d
            filteredQImage = QImage(clonedImage.data, clonedImage.cols, clonedImage.rows, c
lonedImage.step, QImage::Format_ARGB32).copy();
        } else if (outputImage.channels() == 3) { // RGB image
            cv::Mat rgbImage;
            cv::cvtColor(outputImage, rgbImage, cv::COLOR_BGR2RGB); // Convert BGR to RGB
            QImage tempImage(rgbImage.data, rgbImage.cols, rgbImage.rows, rgbImage.step, QI
mage::Format_RGB888);
            filteredQImage = tempImage.copy(); // Copy ensures memory remains valid
        } else if (outputImage.channels() == 1) { // Grayscale image
            QImage tempImage(outputImage.data, outputImage.cols, outputImage.rows, outputIm
age.step, QImage::Format_Grayscale8);
            filteredQImage = tempImage.copy(); // Copy ensures memory remains valid
        } else {
            throw std::runtime_error("Unsupported image format.");
        }

        // Display the filtered image in the QLabel
        ui->FilteredImageLabel->setPixmap(
            QPixmap::fromImage(filteredQImage).scaled(
                ui->FilteredImageLabel->size(),
                Qt::KeepAspectRatio,
                Qt::SmoothTransformation
                )
            );
        ui->FilteredImageLabel->setAlignment(Qt::AlignCenter);

        //QMessageBox::information(this, "Filter Applied", "The filter has been applied suc
cessfully.");
    } catch (const exception& e) {
        QMessageBox::critical(this, "Error", QString("An error occurred: %1").arg(e.what())
);
    }
}

void DescriptorDetails::on_SaveChanges_clicked()
{
    QString libraryPath = this->getLibraryPath();

    qDebug() << "Image Path";
    qDebug() << currentDescriptor->getImage().getPath();

    unsigned int CurrentIdD = currentDescriptor->getIdDescriptor();
    QString appPath = QCoreApplication::applicationDirPath();
```

```
    QJsonObject curObj =  currentDescriptor->toJson();

    // load the library
    qDebug() << "Library to edit";
    qDebug() << libraryPath;
    QFile file( libraryPath);

    if (!file.open(QIODevice::ReadOnly)) {
        qDebug() << "Error: Could not open file";
        return;
    }

    // Read the existing JSON file
    QByteArray data = file.readAll();
    file.close();
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["library"].toArray();
    QJsonArray newArray;

    QString imagePathToDelete;

    for (int i = 0; i < array.size(); i++) {

        QJsonObject obj = array[i].toObject();

        if (obj["id"].toInt() == CurrentIdD) {
            newArray.append(curObj);
        }else{
            newArray.append(obj);
        }

    }
    obj["library"] = newArray;

    if (!file.open(QIODevice::WriteOnly)) {
        qDebug() << "Error: Could not open file";
        return;
    }

    file.write(QJsonDocument(obj).toJson());
    file.close();

    QPixmap pixmap = ui->FilteredImageLabel->pixmap(Qt::ReturnByValue);

    // Check if the pixmap is valid
    if (!pixmap.isNull()) {
        QImage filteredQImage = pixmap.toImage();

        // Open a file dialog to choose the save location
        QString savePath = QFileDialog::getSaveFileName(this, "Save Filtered Image", "", "I
mages (*.png *.jpg *.bmp)");

        // Save the image if a path is provided
        if (!savePath.isEmpty()) {
            if (!savePath.contains('.')) {
                    savePath.append(".png");}

            filteredQImage.save(savePath);
        }
        qDebug() << "Saved to path:"<<savePath;

    } else {
        QMessageBox::warning(this, "Save Error", "No filtered image to save.");
    }
```

```
}

void DescriptorDetails::onLabelClicked(QLabel *clickedLabel) {
    // VÃ©rifiez si une image est chargÃ©e dans le QLabel cliquÃ©
    QPixmap pixmap = clickedLabel->pixmap(Qt::ReturnByValue);

    if (pixmap.isNull()) {
        QMessageBox::warning(this, "Erreur", "Aucune image Ã  afficher.");
        return;
    }

    // CrÃ©ez une nouvelle fenÃªtre pour afficher l'image
    QDialog *imageDialog = new QDialog(this);
    imageDialog->setWindowTitle("Image Agrandie");

    // Configurez un QLabel dans la fenÃªtre
    QLabel *imageLabel = new QLabel(imageDialog);
    imageLabel->setPixmap(pixmap);
    imageLabel->setAlignment(Qt::AlignCenter);
    imageLabel->setScaledContents(true);

    // Ajustez la taille de la fenÃªtre
    imageDialog->resize(800, 600);  // Taille par dÃ©faut
    QVBoxLayout *layout = new QVBoxLayout(imageDialog);
    layout->addWidget(imageLabel);
    imageDialog->setLayout(layout);

    // Affichez la fenÃªtre sans bloquer l'exÃ©cution
    imageDialog->show();
}
```

```
#include "image.hpp"
#include <opencv2/opencv.hpp>
#include <QString>
#include <iostream>
#include <fstream>
#include <QDebug>
#include <QCoreApplication>

using namespace std;
using namespace cv;

// Constructeur de la classe Image : initialise une image Ã  partir d'un chemin donnÃ©.

Image::Image(const QString& imgPath) {
    if (imgPath.isEmpty()) {
        qDebug() << "Error: Image path is empty.";
        return;
    }
    QString appPath = QCoreApplication::applicationDirPath();

    this->path = imgPath;
    qDebug() << "Loading image:" << imgPath;
    loadImage(appPath+imgPath);
}

// Charge une image depuis un chemin donnÃ© et initialise ses propriÃ©tÃ©s.

void Image::loadImage(const QString& imgPath) {
    Mat image = imread(imgPath.toStdString(), IMREAD_COLOR);
    this->content = image;

    if (image.empty()) {
        cerr << "Error while loading the image: " << imgPath.toStdString() << endl;
        exit(1);
    }

    int dot = imgPath.lastIndexOf('.');
    if (dot != -1) {
        this->format = imgPath.mid(dot + 1);
    }
        // Calcule le ratio de compression de l'image.

    this->compressionRatio = calculateCompressionRatio(imgPath);
}

// Retourne le contenu de l'image sous forme d'un objet OpenCV Mat.

Mat Image::getContent() const {
    return this->content;
}

// Calcule le ratio de compression de l'image (taille compressÃ©e / taille non compressÃ©e)
.

double Image::calculateCompressionRatio(const QString& imgPath) const {
    Mat image = cv::imread(imgPath.toStdString(), IMREAD_COLOR);

    if (image.empty()) {
        cerr << "Error reading the image!" << endl;
        return 0.0;
    }
    // Taille non compressÃ©e basÃ©e sur les dimensions et les canaux.

    size_t uncompressedSize = image.rows * image.cols * image.channels();
    // Taille compressÃ©e basÃ©e sur la taille rÃ©elle du fichier.

    ifstream file(imgPath.toStdString(), ios::binary | ios::ate);
    if (!file.is_open()) {
```

```
        cerr << "Error while opening the file to calculate the ratio." << endl;
        return 0.0;
    }

    size_t compressedSize = file.tellg();
    file.close();

    return static_cast<double>(compressedSize) / uncompressedSize;
}

// Affiche l'image dans une fenÃªtre en utilisant OpenCV.

void Image::showImage(const QString& imgPath) const {
    Mat image = imread(imgPath.toStdString(), IMREAD_COLOR);
    imshow("Image", image);
    waitKey(0);
}

// Retourne le format du fichier image (extension).

QString Image::getFormat() const {
    return this->format;
}

// Retourne le chemin actuel de l'image.

QString Image::getPath() const {
    return this->path;
}

// Retourne le ratio de compression de l'image.

double Image::getCompressionRatio() const {
    return this->compressionRatio;
}
// Retourne l'identifiant de l'image.

int Image::getId() const {
    return this->idImage;
}
// Met Ã  jour le chemin de l'image.

void Image::setPath(const QString& newPath) {
    this->path = newPath;
}
// Met Ã  jour l'identifiant de l'image.

void Image::setId(const int newID) {
    this->idImage = newID;
}

// Retourne l'image sous forme de QPixmap (pour l'intÃ©gration avec Qt).

QPixmap Image::getPixmap() const {
    if (content.empty()) {
        return QPixmap();
    }
    // Convertit l'image OpenCV en QImage, puis en QPixmap.

    QImage qImage(content.data, content.cols, content.rows, content.step, QImage::Format_RG
B888);
    return QPixmap::fromImage(qImage.rgbSwapped());
}
```

```
#include "imageproccessing.hpp"
#include "kernels.hpp"
#include <QDebug>
#include <cmath>

ImageProccessing::ImageProccessing() {}

#include <opencv2/opencv.hpp>
#include <stdexcept>
#include <algorithm>

using namespace cv;
using namespace std;

/**
 * @brief Faire pivoter une image d'un angle spÃ©cifiÃ© (0, 90, 180, 270, 360).
 *
 * Cette fonction effectue une rotation de l'image d'entrÃ©e selon l'angle donnÃ©. Elle sup
porte des angles
 * spÃ©cifiques comme 0, 90, 180, 270 et 360 degrÃ©s. Les autres angles sont traitÃ©s en re
tournant l'image d'origine.
 *
 * @param inputImage L'image Ã  faire pivoter.
 * @param angle L'angle de rotation (en degrÃ©s).
 *
 * @return L'image pivotÃ©e de type Mat.
 */
Mat ImageProccessing::rotateImage(const Mat& inputImage, int angle) {

    Mat rotatedImage;

    // GÃ©rer les angles de rotation spÃ©cifiques (0, 90, 180, 360)
    switch (angle) {
        case 90:
            // Rotation de 90 degrÃ©s dans le sens horaire
            transpose(inputImage, rotatedImage); // Transposer (Ã©changer les lignes et les
 colonnes)
            flip(rotatedImage, rotatedImage, 1);  // Retourner horizontalement
            break;
        case 180:
            // Rotation de 180 degrÃ©s
            flip(inputImage, rotatedImage, -1);  // Retourner Ã  la fois horizontalement et
 verticalement
            break;
        case 270:
            // Rotation de 270 degrÃ©s (identique Ã  -90 degrÃ©s)
            transpose(inputImage, rotatedImage); // Transposer (Ã©changer les lignes et les
 colonnes)
            flip(rotatedImage, rotatedImage, 0);  // Retourner verticalement
            break;
        case 360:
            // Rotation de 360 degrÃ©s (aucun changement)
            rotatedImage = inputImage.clone();
            break;
        case 0:
        default:
            // Si 0 degrÃ©s ou tout autre angle, retourner simplement l'image d'origine
            rotatedImage = inputImage.clone();
            break;
    }

    return rotatedImage;
}

/**
 * @brief Applique une convolution sur une image en utilisant un noyau (kernel).
 *
 * Cette fonction effectue une convolution entre une image et un noyau (kernel). Chaque pix
```

```
el de l'image
 * est mis Ã  jour en fonction de la somme pondÃ©rÃ©e des pixels voisins, selon les valeurs
 du noyau.
 * Le rÃ©sultat est stockÃ© dans une nouvelle image.
 *
 * @param kernel Le noyau de convolution, une matrice 2D (vector de vector de float).
 * @param image L'image d'entrÃ©e sur laquelle la convolution est appliquÃ©e.
 *
 * @return Mat L'image de sortie aprÃ¨s convolution, de type CV_64F.
 *
 * @throws runtime_error Si l'image d'entrÃ©e est vide.
 *
 * @note Les valeurs du noyau sont appliquÃ©es sur chaque pixel de l'image, en prenant en c
ompte les pixels voisins
 *       (en utilisant des indices valides Ã  l'intÃ©rieur de l'image). Les bords de l'imag
e sont traitÃ©s en
 *       ignorant les valeurs du noyau qui sortiraient de l'image.
 */
Mat convolution(const vector<vector<float>>& kernel, const Mat& image) {
    if (image.empty()) {
        throw runtime_error("L'image d'entrÃ©e est vide.");
    }

    int kernelWidth = kernel[0].size();
    int kernelHeight = kernel.size();
    int offsetX = kernelWidth / 2;
    int offsetY = kernelHeight / 2;

    // Image de sortie
    Mat output(image.size(), CV_64F);

    // Convolution
    for (int y = 0; y < image.rows; ++y) {
        for (int x = 0; x < image.cols; ++x) {
            float sum = 0.0;

            // Appliquer le noyau (kernel) autour du pixel
            for (int j = 0; j < kernelHeight; ++j) {
                for (int i = 0; i < kernelWidth; ++i) {
                    int imageX = x + i - offsetX;
                    int imageY = y + j - offsetY;

                    // VÃ©rifier si le pixel est Ã  l'intÃ©rieur de l'image
                    if (imageX >= 0 && imageX < image.cols && imageY >= 0 && imageY < image
.rows) {
                        sum += kernel[j][i] * image.at<uchar>(imageY, imageX);
                    }
                }
            }

            // Limiter la valeur Ã  [0, 255]
            output.at<float>(y, x) = sum;
        }
    }

    return output;
}

/**
 * @brief GÃ©nÃ¨re un noyau gaussien de taille et de sigma spÃ©cifiÃ©s.
 *
 * Cette fonction gÃ©nÃ¨re un noyau unidimensionnel de type gaussien. Le noyau est utilisÃ©
 pour des opÃ©rations
 * de filtrage, comme le flou gaussien, et peut Ãªtre utilisÃ© dans des algorithmes de trai
tement d'images.
 * Si certaines conditions sont remplies, un noyau gaussien fixe est utilisÃ© Ã  la place d
u calcul.
 *
```

```
 * @param n La taille du noyau gaussien (doit Ãªtre impair).
 * @param sigma L'Ã©cart-type de la fonction gaussienne. Si ce paramÃ¨tre est nÃ©gatif ou n
ul, une valeur par dÃ©faut
 *              est calculÃ©e en fonction de `n`.
 * @param ktype Le type des Ã©lÃ©ments du noyau, qui peut Ãªtre soit `CV_32F` (32 bits flot
tants) ou `CV_64F` (64 bits flottants).
 *
 * @return Mat Le noyau gaussien gÃ©nÃ©rÃ©, de taille `n` x 1, avec le type spÃ©cifiÃ© par
`ktype`.
 *
 * @throws cv::Exception Si `ktype` n'est pas l'un des types valides (`CV_32F` ou `CV_64F`)
.
 *
 * @note Si `n` est infÃ©rieur ou Ã©gal Ã  une certaine taille (7) et que `sigma` est nul o
u nÃ©gatif,
 *       un noyau prÃ©-calculÃ© est utilisÃ© pour optimiser les performances.
 *
 * @note Le noyau est normalisÃ© de maniÃ¨re Ã  ce que la somme des Ã©lÃ©ments soit Ã©gale
Ã  1.
 */
Mat getGaussianKernel(int n, double sigma, int ktype) {
    const int SMALL_GAUSSIAN_SIZE = 7;
    static const float small_gaussian_tab[][SMALL_GAUSSIAN_SIZE] = {
        {1.f},
        {0.25f, 0.5f, 0.25f},
        {0.0625f, 0.25f, 0.375f, 0.25f, 0.0625f},
        {0.03125f, 0.109375f, 0.21875f, 0.28125f, 0.21875f, 0.109375f, 0.03125f}
    };

    // SÃ©lectionner un noyau fixe si les conditions sont remplies
    const float* fixed_kernel = n % 2 == 1 && n <= SMALL_GAUSSIAN_SIZE && sigma <= 0 ?
                                    small_gaussian_tab[n >> 1] : 0;

    CV_Assert(ktype == CV_32F || ktype == CV_64F);
    Mat kernel(n, 1, ktype);
    float* cf = (float*)kernel.data;
    double* cd = (double*)kernel.data;

    double sigmaX = sigma > 0 ? sigma : ((n - 1) * 0.5 - 1) * 0.3 + 0.8;
    double scale2X = -0.5 / (sigmaX * sigmaX);
    double sum = 0;

    // Calculer les valeurs du noyau
    for (int i = 0; i < n; i++) {
        double x = i - (n - 1) * 0.5;
        double t = fixed_kernel ? (double)fixed_kernel[i] : exp(scale2X * x * x);
        if (ktype == CV_32F) {
            cf[i] = (float)t;
            sum += cf[i];
        } else {
            cd[i] = t;
            sum += cd[i];
        }
    }

    // Normaliser le noyau
    sum = 1. / sum;
    for (int i = 0; i < n; i++) {
        if (ktype == CV_32F)
            cf[i] = (float)(cf[i] * sum);
        else
            cd[i] *= sum;
    }

    return kernel;
}

/**
```

```
 * @brief Applique un flou gaussien Ã  une image en utilisant un noyau gaussien.
 *
 * Cette fonction effectue un flou gaussien sur l'image d'entrÃ©e en utilisant un noyau gau
ssien Ã  deux dimensions.
 * Le noyau est soit spÃ©cifiÃ© par la taille 'ksize', soit calculÃ© automatiquement en fon
ction de la valeur de 'sigma1' et 'sigma2'.
 * La fonction applique la convolution gaussienne sÃ©parÃ©e dans les directions horizontale
 et verticale.
 *
 * @param src L'image d'entrÃ©e, de type 'Mat'.
 * @param dst L'image de sortie, de type 'Mat'. Elle contient l'image aprÃ¨s application du
 flou gaussien.
 * @param ksize La taille du noyau de convolution. Si l'une des dimensions est infÃ©rieure
ou Ã©gale Ã  zÃ©ro, elle sera calculÃ©e automatiquement en fonction de 'sigma1' et 'sigma2'
.
 * @param sigma1 L'Ã©cart-type de la distribution gaussienne dans la direction horizontale.
 Si 'sigma2' est infÃ©rieur ou Ã©gal Ã  zÃ©ro, cette valeur est utilisÃ©e pour les deux dir
ections.
 * @param sigma2 L'Ã©cart-type de la distribution gaussienne dans la direction verticale. S
i cette valeur est infÃ©rieure ou Ã©gale Ã  zÃ©ro, 'sigma1' est utilisÃ©e pour les deux dir
ections.
 * @param borderType Le type de bordure Ã  utiliser pour gÃ©rer les bords de l'image (par e
xemple, 'BORDER_DEFAULT', 'BORDER_REFLECT', etc.).
 *
 * @throws cv::Exception Si les tailles du noyau ('ksize.width' et 'ksize.height') ne sont
pas impaires ou si elles sont invalides.
 *
 * @note Si 'ksize' est dÃ©fini sur '(1, 1)', l'image source est copiÃ©e directement dans l
'image de destination sans appliquer de flou.
 *       Si 'ksize.width' ou 'ksize.height' est infÃ©rieur ou Ã©gal Ã  zÃ©ro, la taille du
noyau est automatiquement calculÃ©e en fonction de 'sigma1' et 'sigma2'.
 *       La fonction applique un filtrage sÃ©parÃ© dans les directions horizontale et verti
cale en utilisant des noyaux gaussiens calculÃ©s avec 'cv::getGaussianKernel'.
 */
void GaussianBlur(const Mat& src, Mat& dst, Size ksize, double sigma1, double sigma2, int b
orderType) {
    if (ksize.width == 1 && ksize.height == 1) {
        src.copyTo(dst);
        return;
    }

    int depth = src.depth();
    if (sigma2 <= 0)
        sigma2 = sigma1;

    // DÃ©tection automatique de la taille du noyau Ã  partir de sigma
    if (ksize.width <= 0 && sigma1 > 0)
        ksize.width = cvRound(sigma1 * (depth == CV_8U ? 3 : 4) * 2 + 1) | 1;
    if (ksize.height <= 0 && sigma2 > 0)
        ksize.height = cvRound(sigma2 * (depth == CV_8U ? 3 : 4) * 2 + 1) | 1;

    CV_Assert(ksize.width > 0 && ksize.width % 2 == 1 &&
              ksize.height > 0 && ksize.height % 2 == 1);

    sigma1 = max(sigma1, 0.);
    sigma2 = max(sigma2, 0.);

    // Obtenir les noyaux gaussiens
    Mat kx = cv::getGaussianKernel(ksize.width, sigma1, CV_32F);
    Mat ky;
    if (ksize.height == ksize.width && abs(sigma1 - sigma2) < DBL_EPSILON)
        ky = kx;
    else
        ky = cv::getGaussianKernel(ksize.height, sigma2, CV_32F);

    // Appliquer la convolution sÃ©parÃ©e
    Mat temp;
    sepFilter2D(src, temp, -1, kx, ky, cv::Point(-1, -1), 0, borderType);
```

```
    dst = temp;
}

/**
 * @brief Convertit une image en niveaux de gris.
 *
 * Cette fonction effectue la conversion d'une image en niveaux de gris en fonction du nomb
re de canaux de l'image d'entrÃ©e.
 * Si l'image a 4 canaux (BGRA), l'alpha est ignorÃ©. Si l'image a 2 canaux, elle est d'abo
rd dupliquÃ©e pour crÃ©er une image Ã  3 canaux.
 * Si l'image est dÃ©jÃ  en niveaux de gris (1 canal), elle est renvoyÃ©e sans modification
.
 *
 * @param inputImage L'image d'entrÃ©e Ã  convertir en niveaux de gris.
 *
 * @return Mat L'image convertie en niveaux de gris, de type `CV_8U` avec un seul canal.
 *
 * @throws runtime_error Si le nombre de canaux de l'image n'est pas pris en charge pour la
 conversion.
 *
 * @note Cette fonction prend en charge les images ayant 1, 2, 3 ou 4 canaux. Si l'image a
2 canaux, ceux-ci sont fusionnÃ©s
 *       pour crÃ©er une image Ã  3 canaux avant la conversion en niveaux de gris.
 *
 * @warning Cette fonction suppose que l'image d'entrÃ©e est dans l'un des formats suivants
 :
 *          - 4 canaux (BGRA) : L'alpha est ignorÃ©.
 *          - 3 canaux (BGR) : Conversion directe en niveaux de gris.
 *          - 2 canaux : Fusion des deux canaux en une image Ã  3 canaux avant conversion.
 *          - 1 canal (Grayscale) : Aucune conversion nÃ©cessaire.
 */
Mat ImageProccessing::toGrayScale(const Mat& inputImage) {

    qDebug() << "DÃ©but de la conversion en niveaux de gris.";
    qDebug() << "Taille de l'image d'entrÃ©e:" << inputImage.cols << "x" << inputImage.rows
;
    qDebug() << "Nombre de canaux de l'image d'entrÃ©e:" << inputImage.channels();
    qDebug() << "Type de l'image d'entrÃ©e:" << inputImage.type();

    Mat grayImage;

    if (inputImage.channels() == 4) {
        qDebug() << "Conversion de BGRA en BGR en supprimant le canal alpha.";
        Mat bgrImage;
        cvtColor(inputImage, bgrImage, COLOR_BGRA2BGR);
        qDebug() << "Conversion de BGR en niveaux de gris.";
        cvtColor(bgrImage, grayImage, COLOR_BGR2GRAY);
    } else if (inputImage.channels() == 3) {
        qDebug() << "Conversion de BGR en niveaux de gris.";
        cvtColor(inputImage, grayImage, COLOR_BGR2GRAY);
    } else if (inputImage.channels() == 2) {
        qDebug() << "Conversion d'une image Ã  2 canaux en niveaux de gris.";

        Mat mergedChannels;
        merge(vector<Mat>{inputImage, inputImage}, mergedChannels);  // Exemple : dupliquer
 pour crÃ©er une image 3 canaux
        cvtColor(mergedChannels, grayImage, COLOR_BGR2GRAY);
    } else if (inputImage.channels() == 1) {
        qDebug() << "L'image est dÃ©jÃ  en niveaux de gris.";
        grayImage = inputImage.clone();
    } else {
        throw runtime_error("Nombre de canaux non supportÃ© pour la conversion en niveaux d
e gris.");
    }

    qDebug() << "Valeur du pixel Ã  (0, 0) dans l'image en niveaux de gris:" << grayImage.a
t<uchar>(0, 0);
```

```
    qDebug() << "Conversion en niveaux de gris terminÃ©e avec succÃ¨s.";

    return grayImage;
}

/**
 * @brief Applique un filtre mÃ©dian personnalisÃ© sur une image (grayscale ou couleur).
 *
 * Cette fonction applique un filtre mÃ©dian sur l'image d'entrÃ©e. Le filtre mÃ©dian rempl
ace chaque pixel par la mÃ©diane
 * des pixels voisins dans une fenÃªtre de taille `kernelSize`. Si l'image est en niveaux d
e gris, le filtre est appliquÃ©
 * directement sur l'image. Si l'image est en couleur, le filtre est appliquÃ© sÃ©parÃ©ment
 sur chaque canal (R, G, B).
 *
 * @param inputImage L'image d'entrÃ©e Ã  laquelle le filtre mÃ©dian sera appliquÃ©. Elle p
eut Ãªtre en niveaux de gris ou en couleur.
 * @param kernelSize La taille du noyau du filtre mÃ©dian. Ce paramÃ¨tre doit Ãªtre un nomb
re impair et supÃ©rieur ou Ã©gal Ã  3.
 *
 * @return Mat L'image filtrÃ©e aprÃ¨s application du filtre mÃ©dian.
 *
 * @throws std::invalid_argument Si `kernelSize` est un nombre pair ou infÃ©rieur Ã  3.
 * @throws std::runtime_error Si l'image d'entrÃ©e est vide.
 *
 * @note Le filtre mÃ©dian est appliquÃ© sÃ©parÃ©ment pour chaque pixel en fonction de ses
voisins dans un voisinage dÃ©fini par `kernelSize`.
 *       Les bords de l'image sont gÃ©rÃ©s en utilisant la fonction `clamp` pour Ã©viter l'
accÃ¨s Ã  des pixels en dehors de l'image.
 *
 * @warning Cette fonction fonctionne Ã  la fois pour des images en niveaux de gris (1 cana
l) et en couleur (3 canaux).
 */
Mat ImageProccessing::applyCustomMedianFilter(const Mat& inputImage, int kernelSize) {

    if (kernelSize % 2 == 0 || kernelSize < 3) {
        throw invalid_argument("La taille du noyau doit Ãªtre un nombre impair et >= 3");
    }

      // VÃ©rifier si l'image est vide
    if (inputImage.empty()) {
        throw runtime_error("L'image d'entrÃ©e est vide. Impossible d'appliquer le filtre."
);
    }

    if (inputImage.channels() == 1) {
        // Initialiser l'image de sortie avec la mÃªme taille et le mÃªme type
        Mat filteredImage = Mat::zeros(inputImage.size(), CV_8UC1);

        int halfKernel = kernelSize / 2;

        // Appliquer le filtre mÃ©dian
        for (int y = 0; y < inputImage.rows; ++y) {
            for (int x = 0; x < inputImage.cols; ++x) {
                vector<uchar> neighborhood;

                for (int dy = -halfKernel; dy <= halfKernel; ++dy) {
                    for (int dx = -halfKernel; dx <= halfKernel; ++dx) {
                        int ny = clamp(y + dy, 0, inputImage.rows - 1);
                        int nx = clamp(x + dx, 0, inputImage.cols - 1);
                        neighborhood.push_back(inputImage.at<uchar>(ny, nx));
                    }
                }

                // Trier les valeurs du voisinage et sÃ©lectionner la mÃ©diane
                sort(neighborhood.begin(), neighborhood.end());
                filteredImage.at<uchar>(y, x) = neighborhood[neighborhood.size() / 2];
            }
```

```
        }

        return filteredImage;
    }

    // Initialiser les canaux de sortie
    vector<Mat> channels;
    split(inputImage, channels); // Diviser l'image couleur en 3 canaux (B, G, R)

    // Appliquer le filtre mÃ©dian personnalisÃ© sur chaque canal
    for (size_t i = 0; i < channels.size(); ++i) {
        Mat filteredChannel = Mat::zeros(channels[i].size(), CV_8UC1);

        int halfKernel = kernelSize / 2;
        for (int y = 0; y < channels[i].rows; ++y) {
            for (int x = 0; x < channels[i].cols; ++x) {
                vector<uchar> neighborhood;

                for (int dy = -halfKernel; dy <= halfKernel; ++dy) {
                    for (int dx = -halfKernel; dx <= halfKernel; ++dx) {
                        int ny = clamp(y + dy, 0, channels[i].rows - 1);
                        int nx = clamp(x + dx, 0, channels[i].cols - 1);
                        neighborhood.push_back(channels[i].at<uchar>(ny, nx));
                    }
                }

                sort(neighborhood.begin(), neighborhood.end());
                filteredChannel.at<uchar>(y, x) = neighborhood[neighborhood.size() / 2];
            }
        }

        // Remplacer le canal par le rÃ©sultat filtrÃ©
        channels[i] = filteredChannel.clone();
    }

    // Combiner les canaux filtrÃ©s pour recrÃ©er l'image couleur
    Mat filteredImage;
    merge(channels, filteredImage);

    return filteredImage;
}

/**
 * @brief Applique la dÃ©tection des contours en utilisant l'opÃ©rateur Sobel.
 *
 * Cette fonction utilise les noyaux de Sobel pour dÃ©tecter les contours dans une image en
 calculant les gradients dans les directions horizontale et verticale. Ensuite, elle combin
e ces gradients pour obtenir la magnitude des contours, rÃ©sultant en une image binaire rep
rÃ©sentant les bords dÃ©tectÃ©s.
 *
 * @param image L'image d'entrÃ©e en niveaux de gris (de type `CV_8UC1`) sur laquelle la dÃ
©tection des contours sera effectuÃ©e.
 * @return Mat L'image rÃ©sultante contenant les contours dÃ©tectÃ©s. Elle est de mÃªme tai
lle que l'image d'entrÃ©e et est de type `CV_8UC1`.
 *
 * @note L'opÃ©rateur Sobel est utilisÃ© pour calculer les dÃ©rivÃ©es de l'intensitÃ© dans
les directions horizontale et verticale, puis ces dÃ©rivÃ©es sont combinÃ©es pour calculer
la magnitude des gradients.
 * @note L'image rÃ©sultante est une image en niveaux de gris oÃ¹ les pixels blancs (valeur
 proche de 255) reprÃ©sentent les contours dÃ©tectÃ©s et les pixels noirs (valeur proche de
 0) reprÃ©sentent les zones sans contours.
 *
 * @warning Cette fonction suppose que l'image d'entrÃ©e est en niveaux de gris (1 canal).
Si l'image est en couleur ou a plus de 1 canal, un prÃ©traitement est nÃ©cessaire.
 */
Mat ImageProccessing::applyEdgeDetection(const Mat& Inputimage) {

    if (Inputimage.empty()) {
```

```
        throw runtime_error("L'image d'entrÃ©e est vide. Impossible d'appliquer le traiteme
nt.");
    }

    Mat grayImage = toGrayScale(Inputimage);

    qDebug() << "Type de l'image aprÃ¨s conversion en niveaux de gris:" << grayImage.type()
;
    qDebug() << "Taille de l'image: largeur =" << grayImage.size().width << ", hauteur =" <
< grayImage.size().height;

    // DÃ©finir les noyaux de Sobel
    vector<vector<float>> sobelX = {
        {-1, 0, 1},
        {-2, 0, 2},
        {-1, 0, 1}
    };

    vector<vector<float>> sobelY = {
        {-1, -2, -1},
        { 0,  0,  0},
        { 1,  2,  1}
    };

    // Calculer les gradients en X et Y
    Mat gradX = convolution(sobelX, grayImage);
    Mat gradY = convolution(sobelY, grayImage);

    // Calculer la magnitude du gradient
    Mat edges = Mat::zeros(grayImage.size(), CV_8UC1);
    for (int y = 0; y < grayImage.rows; ++y) {
        for (int x = 0; x < grayImage.cols; ++x) {
            float gx = gradX.at<float>(y, x); // AccÃ©der en tant que float
            float gy = gradY.at<float>(y, x); // AccÃ©der en tant que float
            float magnitude = sqrt(gx * gx + gy * gy);

            // Normaliser les valeurs entre [0, 255]
            edges.at<uchar>(y, x) = static_cast<uchar>(clamp(magnitude, 0.0f, 255.0f));
        }
    }

    qDebug() << "DÃ©tection des contours terminÃ©e avec succÃ¨s.";

    return edges;
}

/**
 * @brief Applique un seuillage simple sur une image en niveaux de gris.
 *
 * Cette fonction prend une image en entrÃ©e, la convertit en niveaux de gris si elle est e
n couleur,
 * puis applique un seuillage simple pour crÃ©er une image binaire. Tous les pixels ayant u
ne intensitÃ©
 * supÃ©rieure Ã  la valeur seuil sont dÃ©finis comme blancs (255), tandis que les autres s
ont dÃ©finis comme noirs (0).
 *
 * @param inputImage L'image d'entrÃ©e (peut Ãªtre en couleur ou en niveaux de gris).
 *
 * @return Mat L'image binaire rÃ©sultante aprÃ¨s application du seuillage.
 *
 * @note Si l'image d'entrÃ©e est en couleur, elle est convertie en niveaux de gris avant d
'appliquer le seuillage.
 *       La valeur de seuil est fixÃ©e Ã  128 dans cette implÃ©mentation.
 */
Mat ImageProccessing::applyThreshold(const Mat& inputImage, int thresholdValue) {

    // Convertir l'image en niveaux de gris si elle est en couleur
    Mat grayImage;
```

```
    if (inputImage.channels() > 1) {
        grayImage = toGrayScale(inputImage);
    } else {
        grayImage = inputImage;
    }

    // CrÃ©er une image binaire pour stocker le rÃ©sultat du seuillage
    Mat binaryImage = Mat::zeros(grayImage.size(), CV_8UC1);

    // Appliquer le seuillage
    for (int y = 0; y < grayImage.rows; y++) {
        for (int x = 0; x < grayImage.cols; x++) {
            // Obtenir l'intensitÃ© du pixel
            uchar pixelValue = grayImage.at<uchar>(y, x);

            // Appliquer le seuil
            if (pixelValue > thresholdValue) {
                binaryImage.at<uchar>(y, x) = 255; // Pixel blanc (objet)
            } else {
                binaryImage.at<uchar>(y, x) = 0;   // Pixel noir (fond)
            }
        }
    }

    qDebug() << "Seuillage terminÃ© avec succÃ¨s.";

    return binaryImage;
}

/**
 * @brief Calcule l'histogramme d'une image et retourne l'image de l'histogramme.
 *
 * Cette fonction effectue les Ã©tapes suivantes :
 * - Conversion de l'image en niveaux de gris si l'image est en couleur (RGB).
 * - Calcul de l'histogramme des intensitÃ©s de pixels.
 * - Normalisation de l'histogramme pour le redimensionner Ã  une taille d'image spÃ©cifiÃ©
e.
 * - Dessin de l'histogramme normalisÃ© sur une image de fond blanc.
 *
 * @param inputImage L'image d'entrÃ©e pour laquelle l'histogramme doit Ãªtre calculÃ©. Ell
e peut Ãªtre en couleur ou en niveaux de gris.
 * @return Mat L'image reprÃ©sentant l'histogramme normalisÃ© de l'image d'entrÃ©e.
 *
 * @note Cette fonction fonctionne uniquement avec des images en niveaux de gris ou en coul
eur (3 canaux).
 * Si l'image est en couleur, elle sera convertie en niveaux de gris avant le calcul de l'h
istogramme.
 *
 * @warning L'image d'entrÃ©e doit Ãªtre une image valide, sinon un comportement indÃ©fini
pourrait se produire.
 */
Mat ImageProccessing::calculateHistogram(const Mat& inputImage) {
    // Ã\211tape 1 : Conversion en niveaux de gris si nÃ©cessaire
    Mat grayImage;
    if (inputImage.channels() > 1) {
        cvtColor(inputImage, grayImage, COLOR_BGR2GRAY);
    } else {
        grayImage = inputImage.clone();
    }

    // Ã\211tape 2 : Initialiser un tableau pour l'histogramme (256 bins)
    int histSize = 256;
    vector<int> histogram(histSize, 0);

    // Ã\211tape 3 : Parcourir l'image pour remplir l'histogramme
    for (int y = 0; y < grayImage.rows; y++) {
        for (int x = 0; x < grayImage.cols; x++) {
```

```
            int pixelValue = grayImage.at<uchar>(y, x);
            histogram[pixelValue]++;
        }
    }

    // Ã\211tape 4 : Normalisation de l'histogramme pour le redimensionner Ã  l'image
    int maxVal = *max_element(histogram.begin(), histogram.end());

    if (maxVal == 0) {
        return Mat(); // Retourner une image vide en cas de problÃ¨me
    }

    int histWidth = 512, histHeight = 400;
    Mat histImage(histHeight + 50, histWidth + 50, CV_8UC3, Scalar(255, 255, 255)); // Fond
 blanc

    // Normalisation pour que les valeurs soient entre 0 et histHeight
    for (int i = 0; i < histSize; i++) {
        histogram[i] = ((double)histogram[i] / maxVal) * histHeight;
    }

    // Ã\211tape 5 : Dessin de l'histogramme
    int binWidth = cvRound((double)histWidth / histSize);
    Scalar barColor = Scalar(50, 50, 150); // Bleu foncÃ© pour les barres
    for (int i = 0; i < histSize; i++) {
        rectangle(histImage,
                  Point(25 + binWidth * i, histHeight + 25),
                  Point(25 + binWidth * (i + 1), histHeight + 25 - histogram[i]),
                  barColor, FILLED);
    }

    // Ajouter les axes
    line(histImage, Point(25, 25), Point(25, histHeight + 25), Scalar(0, 0, 0), 2); // Axe
vertical
    line(histImage, Point(25, histHeight + 25), Point(histWidth + 25, histHeight + 25), Sca
lar(0, 0, 0), 2); // Axe horizontal

    // Ajouter des annotations pour les axes
    putText(histImage, "Intensite", Point(histWidth / 2, histHeight + 45), FONT_HERSHEY_SIM
PLEX, 0.6, Scalar(0, 0, 0), 1);
    putText(histImage, "0", Point(20, histHeight + 30), FONT_HERSHEY_SIMPLEX, 0.5, Scalar(0
, 0, 0), 1);
    putText(histImage, "255", Point(histWidth, histHeight + 30), FONT_HERSHEY_SIMPLEX, 0.5,
 Scalar(0, 0, 0), 1);

    // Ajouter des valeurs sur l'axe des ordonnÃ©es (par exemple : 0, max/2, max)
    putText(histImage, "0", Point(5, histHeight + 25), FONT_HERSHEY_SIMPLEX, 0.5, Scalar(0,
 0, 0), 1);
    putText(histImage, to_string(maxVal / 2), Point(5, (histHeight + 25) / 2), FONT_HERSHEY
_SIMPLEX, 0.5, Scalar(0, 0, 0), 1);
    putText(histImage, to_string(maxVal), Point(5, 25), FONT_HERSHEY_SIMPLEX, 0.5, Scalar(0
, 0, 0), 1);

    return histImage;
}

/**
 * @brief Applique l'algorithme SIFT (Scale-Invariant Feature Transform) pour dÃ©tecter des
 points-clÃ©s
 *        et descripteurs d'une image.
 *
 * Cette fonction implÃ©mente SIFT, un algorithme robuste utilisÃ© pour dÃ©tecter des carac
tÃ©ristiques
 * distinctives dans une image, mÃªme sous des variations d'Ã©chelle, de rotation et d'illu
mination.
 *
 * ### Fonctionnement :
 * - Convertit l'image en niveaux de gris (si nÃ©cessaire) pour simplifier le traitement.
```

```
 * - DÃ©tecte des points-clÃ©s dans l'image, qui correspondent Ã  des rÃ©gions d'intÃ©rÃªt
importantes.
 * - GÃ©nÃ¨re des descripteurs pour chaque point-clÃ©, reprÃ©sentant les caractÃ©ristiques
locales.
 * - Superpose les points-clÃ©s dÃ©tectÃ©s sur l'image d'entrÃ©e pour visualiser les rÃ©sul
tats.
 *
 * ### Utilisations :
 * - Correspondance d'images (image matching).
 * - Suivi d'objets (object tracking).
 * - Reconstruction 3D basÃ©e sur des images.
 * - Classification et reconnaissance d'objets.
 *
 * @param inputImage L'image d'entrÃ©e pour laquelle les points-clÃ©s et descripteurs seron
t calculÃ©s.
 *                   Elle peut Ãªtre en couleur ou en niveaux de gris.
 * @return cv::Mat Une copie de l'image d'entrÃ©e avec les points-clÃ©s superposÃ©s en vert
.
 * @throws std::invalid_argument Si l'image d'entrÃ©e est vide.
 *
 * @note Cette implÃ©mentation utilise la classe `cv::SIFT` fournie par OpenCV.
 *
 * ### Exemple d'utilisation :
 * @code
 * cv::Mat image = cv::imread("exemple.jpg"); // Charger une image
 * ImageProcessing processor;
 * cv::Mat resultat = processor.appliquerSIFT(image);
 * cv::imshow("RÃ©sultat SIFT", resultat); // Afficher l'image avec les points-clÃ©s
 * cv::waitKey(0);
 * @endcode
 */
Mat ImageProccessing::applySIFT(const Mat& inputImage) {
    // Conversion en niveaux de gris (si nÃ©cessaire)
    Mat imageGris;
    if (inputImage.channels() == 3)
        cvtColor(inputImage, imageGris, COLOR_BGR2GRAY);
    else
        imageGris = inputImage;

    // DÃ©tecter les points-clÃ©s et les descripteurs avec SIFT
    Ptr<SIFT> sift = SIFT::create();
    vector<KeyPoint> pointsCles;
    Mat descripteurs;

    sift->detectAndCompute(imageGris, noArray(), pointsCles, descripteurs);

    // Dessiner les points-clÃ©s sur l'image
    Mat imageResultat;
    drawKeypoints(inputImage, pointsCles, imageResultat, Scalar(0, 255, 0));

    return imageResultat;
}

/**
 * @brief Applique l'opÃ©ration d'Ã©rosion sur une image en niveaux de gris.
 *
 * L'Ã©rosion est une opÃ©ration morphologique qui remplace chaque pixel par la valeur mini
male de ses voisins
 * dans un voisinage dÃ©fini par un noyau de taille `kernelSize`. Cette fonction applique l
'Ã©rosion Ã  une image
 * en niveaux de gris. Si l'image d'entrÃ©e est en couleur, elle est d'abord convertie en n
iveaux de gris.
 *
 * @param inputImage L'image d'entrÃ©e sur laquelle l'Ã©rosion sera appliquÃ©e. Elle peut Ã
ªtre en couleur ou en niveaux de gris.
 * @param kernelSize La taille du noyau d'Ã©rosion. Ce paramÃ¨tre doit Ãªtre un entier impa
ir et supÃ©rieur Ã  zÃ©ro.
 *
```

```
 * @return Mat L'image rÃ©sultante aprÃ¨s application de l'Ã©rosion.
 *
 * @throws std::invalid_argument Si `kernelSize` n'est pas un entier positif impair.
 *
 * @note L'Ã©rosion est effectuÃ©e en parcourant chaque pixel de l'image et en remplaÃ§ant
sa valeur par la valeur
 *       minimale de ses voisins dans un voisinage de taille `kernelSize`. Les bords de l'i
mage sont traitÃ©s
 *       en ajoutant des bordures rÃ©flÃ©chissantes pour Ã©viter les artefacts.
 *
 * @warning Cette fonction suppose que l'image est en niveaux de gris ou qu'elle peut Ãªtre
 convertie en niveaux de gris.
 */
Mat ImageProccessing::applyErosion(const Mat& inputImage, int kernelSize) {
    // VÃ©rification : la taille du noyau doit Ãªtre impaire et supÃ©rieure Ã  zÃ©ro
    if (kernelSize <= 0 || kernelSize % 2 == 0) {
        throw invalid_argument("La taille du noyau doit Ãªtre un entier positif impair.");
    }

    // VÃ©rifier si l'image est en niveaux de gris
    Mat grayImage;
    if (inputImage.channels() > 1) {
        cvtColor(inputImage, grayImage, COLOR_BGR2GRAY);
    } else {
        grayImage = inputImage.clone();
    }

    // Ajouter des bordures pour Ã©viter les artefacts sur les bords
    int offset = kernelSize / 2;
    Mat paddedImage;
    copyMakeBorder(grayImage, paddedImage, offset, offset, offset, offset, BORDER_REFLECT);

    // CrÃ©er une image de sortie
    Mat outputImage = Mat::zeros(grayImage.size(), CV_8U);

    // Appliquer l'Ã©rosion manuelle
    for (int i = offset; i < paddedImage.rows - offset; i++) {
        for (int j = offset; j < paddedImage.cols - offset; j++) {
            int minVal = 255;

            // Parcourir le noyau structurant
            for (int ki = -offset; ki <= offset; ki++) {
                for (int kj = -offset; kj <= offset; kj++) {
                    minVal = min(minVal, (int)paddedImage.at<uchar>(i + ki, j + kj));
                }
            }

            outputImage.at<uchar>(i - offset, j - offset) = minVal;
        }
    }

    return outputImage;
}

Mat hardcodedGaussianKernel() {
    return (Mat_<float>(3, 3) <<
                0.0751136, 0.123841, 0.0751136,
            0.123841,  0.204180, 0.123841,
            0.0751136, 0.123841, 0.0751136
            );
}

/**
 * @brief Applique un flou gaussien sur une image Ã  un seul canal Ã  l'aide d'un noyau de
convolution.
 *
 * Cette fonction applique un flou gaussien sur une image en utilisant un noyau de convolut
ion spÃ©cifiÃ©.
```

```
 * Le flou gaussien est une opÃ©ration de filtrage qui remplace chaque pixel de l'image par
 une moyenne pondÃ©rÃ©e
 * de ses voisins, avec des poids donnÃ©s par le noyau gaussien. La fonction est appliquÃ©e
 sur une image Ã  un seul canal
 * (par exemple une image en niveaux de gris).
 *
 * @param src L'image d'entrÃ©e Ã  un seul canal (par exemple en niveaux de gris) sur laque
lle le flou gaussien sera appliquÃ©.
 * @param dst L'image de sortie dans laquelle l'image floutÃ©e sera stockÃ©e.
 * @param kernel Le noyau de convolution Ã  appliquer pour le flou gaussien. Le noyau doit
Ãªtre une matrice de type 'float' de dimensions impaires.
 *
 * @return void Cette fonction ne renvoie rien. Le rÃ©sultat est stockÃ© dans l'image de so
rtie 'dst'.
 *
 * @note Cette fonction effectue la convolution de l'image 'src' avec le noyau spÃ©cifiÃ©.
Elle utilise un format Ã  virgule flottante
 *       pour les calculs intermÃ©diaires afin d'Ã©viter la perte de prÃ©cision, puis norma
lise et convertit le rÃ©sultat final
 *       dans l'Ã©chelle des valeurs d'intensitÃ© [0, 255].
 *
 * @warning L'image d'entrÃ©e 'src' doit Ãªtre une image Ã  un seul canal, et le noyau 'ker
nel' doit Ãªtre de taille impair.
 */
void GaussianBlurSingleChannel(const Mat& src, Mat& dst, const Mat& kernel) {
    int kernelRadius = kernel.rows / 2;

    // CrÃ©er une image intermÃ©diaire pour le rÃ©sultat
    Mat temp = Mat::zeros(src.size(), CV_32F); // Utiliser float pour les calculs intermÃ©d
iaires

    // Effectuer la convolution
    for (int y = 0; y < src.rows; y++) {
        for (int x = 0; x < src.cols; x++) {
            float sum = 0.0;

            for (int j = -kernelRadius; j <= kernelRadius; j++) {
                for (int i = -kernelRadius; i <= kernelRadius; i++) {
                    int nx = clamp(x + i, 0, src.cols - 1); // GÃ©rer les bords
                    int ny = clamp(y + j, 0, src.rows - 1);
                    sum += src.at<uchar>(ny, nx) * kernel.at<float>(j + kernelRadius, i + k
ernelRadius);
                }
            }
            temp.at<float>(y, x) = sum; // Stocker le rÃ©sultat en tant que float
        }
    }

    // Normaliser et reconvertir en format 8 bits
    normalize(temp, temp, 0, 255, NORM_MINMAX); // Normaliser le rÃ©sultat dans l'intervall
e [0, 255]
    temp.convertTo(dst, CV_8UC1); // Convertir en 8 bits unsigned char
}

/**
 * @brief Applique un flou gaussien Ã  une image Ã  plusieurs canaux (couleur ou niveau de
gris).
 *
 * Cette fonction applique un flou gaussien Ã  une image en utilisant un noyau de convoluti
on gaussienne. Elle peut
 * traiter des images en niveaux de gris (1 canal) ou en couleur (3 canaux). Pour une image
 couleur, le flou gaussien
 * est appliquÃ© indÃ©pendamment Ã  chaque canal (R, G, B) de l'image. Si l'image est en ni
veaux de gris, le flou est appliquÃ©
 * directement sur l'image Ã  un seul canal.
 *
 * @param src L'image d'entrÃ©e (en couleur ou en niveaux de gris) Ã  laquelle le flou gaus
sien sera appliquÃ©.
```

```
 * @param dst L'image de sortie dans laquelle l'image floutÃ©e sera stockÃ©e.
 *
 * @return void Cette fonction ne renvoie rien. Le rÃ©sultat est stockÃ© dans l'image `dst`
.
 *
 * @throws std::runtime_error Si l'image d'entrÃ©e `src` n'a pas 1 ou 3 canaux.
 *
 * @note La fonction applique le flou gaussien Ã  chaque canal de l'image individuellement
si l'image est en couleur,
 *       ou directement sur l'image si elle est en niveaux de gris. Un noyau de convolution
 gaussienne durcodÃ© est utilisÃ©
 *       pour le flou. AprÃ¨s application du filtre, les canaux sont recombinÃ©s pour resta
urer l'image en couleur (si applicable).
 *
 * @warning L'image d'entrÃ©e doit Ãªtre une image Ã  1 ou 3 canaux. Si l'image d'entrÃ©e a
 un nombre de canaux diffÃ©rent, une exception sera levÃ©e.
 */
void GaussianBlurMultiChannel(const cv::Mat& src, cv::Mat& dst) {
    // Valider le type d'entrÃ©e (image en niveaux de gris ou en couleur)
    if (src.channels() != 3 && src.channels() != 1) {
        throw runtime_error("L'image d'entrÃ©e doit Ãªtre une image Ã  3 canaux (couleur) o
u 1 canal (niveaux de gris).");
    }

    // Obtenir le noyau gaussien durcodÃ©
    Mat kernel = hardcodedGaussianKernel();

    if(src.channels() == 3){
        // Diviser l'image en ses canaux individuels
        vector<cv::Mat> channels;
        split(src, channels);

        // Appliquer le flou gaussien Ã  chaque canal
        for (int i = 0; i < channels.size(); ++i) {
            cv::Mat temp;
            GaussianBlurSingleChannel(channels[i], temp, kernel); // Appliquer le filtre Ã
 chaque canal
            channels[i] = temp; // Stocker le rÃ©sultat dans le vecteur de canaux
        }
        merge(channels, dst);
    }
    if(src.channels() == 1){
        Mat temp;
        GaussianBlurSingleChannel(src, temp, kernel);
        merge(temp, dst);
    }
}

/**
 * @brief Applique un filtre gaussien Ã  une image d'entrÃ©e.
 *
 * Cette fonction applique un filtre gaussien Ã  l'image d'entrÃ©e en utilisant la fonction
 `GaussianBlurMultiChannel`.
 * Le filtre est appliquÃ© pour lisser l'image, rÃ©duire le bruit ou produire un effet de f
lou. La fonction vÃ©rifie d'abord
 * que l'image d'entrÃ©e n'est pas vide avant d'appliquer le filtre.
 *
 * @param inputImage L'image d'entrÃ©e sur laquelle le filtre gaussien sera appliquÃ©. Cett
e image peut Ãªtre en niveaux de gris
 *                   ou en couleur (avec 1 ou 3 canaux).
 *
 * @return Mat L'image de sortie aprÃ¨s application du filtre gaussien.
 *
 * @throws std::runtime_error Si l'image d'entrÃ©e est vide.
 *
 * @note Cette fonction utilise la mÃ©thode `GaussianBlurMultiChannel` pour appliquer le fi
ltre gaussien. Elle supporte Ã  la fois
 *       les images en niveaux de gris et en couleur.
```

```
 *
 * @warning L’image d’entrÃ©e ne doit pas Ãªtre vide.
 */
Mat ImageProccessing::applyGaussianFilter(const Mat& inputImage) {
    // Valider l’image d’entrÃ©e
    if (inputImage.empty()) {
        throw runtime_error("L’image d’entrÃ©e est vide. Impossible d’appliquer le filtre g
aussien.");
    }

    // Initialiser l’image de sortie
    Mat outputImage;

    // Appliquer le flou gaussien en utilisant l’implÃ©mentation personnalisÃ©e
    GaussianBlurMultiChannel(inputImage, outputImage);

    return outputImage;
}
```

```
#include "librarymanagement.hpp"
#include "descriptor.hpp"
#include <QDebug>
#include <QFile>
#include <QJsonDocument>
#include <QJsonObject>
#include <QJsonArray>
#include <QIODevice>
#include <QTextStream>
#include <QJsonObject>
#include <QInputDialog>
#include <QMessageBox>
#include <QPushButton>
#include <QVBoxLayout>
#include <QLabel>
#include <QCoreApplication>
#include <QDir>

ManageLibrary::ManageLibrary(int acces, Descriptor* head,QString libraryPath): acces(acces)
, head(head) , libraryPath(libraryPath) {};

Descriptor* ManageLibrary::getDescriptor(unsigned int idDesc) const {
    Descriptor* current = head;

    while(current != nullptr) {
        if(current->getIdDescriptor() == idDesc){
            return current;
        }
        current = current->getNextDescriptor();
    }

    //cout << "Error: Descriptor with ID" << idDesc << "not found." << endl ;
    return nullptr;
}

int ManageLibrary::getAcces() const {return this->acces;}

void ManageLibrary::addDescriptor() const {

}

void ManageLibrary::deleteDescriptor() const {}

Descriptor* ManageLibrary::searchDescriptor(unsigned int id) const {

    Descriptor* current = head;

    while(current != nullptr) {
        if(current->getIdDescriptor() == id){
            return current;
        }
        current = current->getNextDescriptor();
    }

    //cout << "Error: Image with ID" << id << "not found." <<endl ;
    return nullptr;

}

int ManageLibrary::totalDescriptors() const {

    int count = 0;
    Descriptor* current = head;
    while(current != nullptr){
        ++count;
        current = current->getNextDescriptor();
    }
```

```
    return count;
}

double ManageLibrary::displayCost(unsigned int id) const {

    Descriptor* current = head;

    while(current != nullptr) {
        if(current->getIdDescriptor() == id){
            return current->getCost();
        }
        current = current->getNextDescriptor();
    }

    // cout << "Error: Image with ID" << id << "not found." <<endl ;
    return -1.0;

}

Descriptor* ManageLibrary::getHead() const {
    return head;
}
void ManageLibrary::display() const {

    Descriptor* current = this->getHead();
    int i = 1;

    qDebug() << "Displaying ... ";
    while(current != nullptr){
        qDebug() << "Desc = " << i;
        current->display();
        current = current->getNextDescriptor();
        i++;
    }


}

double ManageLibrary::getMaxDescriptorCost() {
    double maxCost = 0.0;
    Descriptor* current = this->getHead();
    while (current != nullptr) {
        if (current->getCost() > maxCost) {
            maxCost = current->getCost();
        }
        current = current->getNextDescriptor();
    }
    return maxCost;
}

double ManageLibrary::getMinDescriptorCost() {
    double minCost = INFINITY;
    Descriptor* current = this->getHead();
    while (current != nullptr) {
        if (current->getCost() < minCost) {
            minCost = current->getCost();
        }
        current = current->getNextDescriptor();
    }
    return minCost;
}

void ManageLibrary::deleteDescriptor(Descriptor* descriptorToDelete) {
    if (!descriptorToDelete) {
        qDebug() << "The descriptor to delete is null. Operation aborted.";
        return;
    }
```

```
    qDebug() << "Deleting descriptor: " << descriptorToDelete->getIdDescriptor();

    QFile file(libraryPath);
    if (!file.open(QIODevice::ReadOnly)) {
        qDebug() << "Error: Could not open file";
        return;
    }
    QString appPath = QCoreApplication::applicationDirPath();

    // Read the existing JSON file
    QByteArray data = file.readAll();
    file.close();
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["library"].toArray();
    QJsonArray newArray;

    QString imagePathToDelete;
    for (int i = 0; i < array.size(); i++) {
        QJsonObject obj = array[i].toObject();
        if (obj["id"].toInt() != descriptorToDelete->getIdDescriptor()) {
            newArray.append(obj);
        } else {
            imagePathToDelete = appPath +  obj["Imagepath"].toString();
        }
    }
    obj["library"] = newArray;

    if (!file.open(QIODevice::WriteOnly)) {
        qDebug() << "Error: Could not open file";
        return;
    }
    file.write(QJsonDocument(obj).toJson());
    file.close();

    // Delete the image file associated with the descriptor
    if (!imagePathToDelete.isEmpty()) {
        QFile imageFile(imagePathToDelete);
        if (imageFile.exists()) {
            if (!imageFile.remove()) {
                qDebug() << "Error: Could not delete image file";
            } else {
                qDebug() << "Image file deleted successfully";
            }
        } else {
            qDebug() << "Image file does not exist";
        }
    }

    // Remove the descriptor from the in-memory library object
    Descriptor* current = head;
    Descriptor* previous = nullptr;

    while (current != nullptr) {
        if (current->getIdDescriptor() == descriptorToDelete->getIdDescriptor()) {
            if (previous == nullptr) {
                // The descriptor to delete is the head of the list
                head = current->getNextDescriptor();
            } else {
                previous->setNextDescriptor(current->getNextDescriptor());
            }
            delete current;
            qDebug() << "Descriptor removed from in-memory library";
            return;
        }
        previous = current;
        current = current->getNextDescriptor();
    }
```

```
    qDebug() << "Descriptor not found in in-memory library";
}

ManageLibrary ManageLibrary::orderDescriptorsByCostDescending() {
    // Create a new library to hold the ordered descriptors
    ManageLibrary orderedLibrary = ManageLibrary(0, nullptr,"");

    // Traverse the current library and insert each descriptor into the new library in sort
ed order
    Descriptor* current = head;
    while (current != nullptr) {
        Descriptor* next = current->getNextDescriptor();
        insertDescriptorInOrder(orderedLibrary, current);
        current = next;
    }

    // Return the ordered library
    return orderedLibrary;
}


// Helper function to insert a descriptor into the new library in sorted order
void ManageLibrary::insertDescriptorInOrder(ManageLibrary& library, Descriptor* descriptor)
 {
    if (library.head == nullptr || library.head->getCost() < descriptor->getCost()) {
        // Insert at the beginning
        descriptor->setNextDescriptor(library.head);
        library.head = descriptor;
    } else {
        // Traverse the library to find the correct position
        Descriptor* current = library.head;
        while (current->getNextDescriptor() != nullptr && current->getNextDescriptor()->get
Cost() >= descriptor->getCost()) {
            current = current->getNextDescriptor();
        }
        descriptor->setNextDescriptor(current->getNextDescriptor());
        current->setNextDescriptor(descriptor);
    }
}
ManageLibrary ManageLibrary::orderDescriptorsByCostAscending() {
    // Create a new library to hold the ordered descriptors
    ManageLibrary orderedLibrary = ManageLibrary(0, nullptr,"");

    // Traverse the current library and insert each descriptor into the new library in sort
ed order
    Descriptor* current = head;
    while (current != nullptr) {
        Descriptor* next = current->getNextDescriptor();
        insertDescriptorInOrderAscending(orderedLibrary, current);
        current = next;
    }

    // Return the ordered library
    return orderedLibrary;
}

// Helper function to insert a descriptor into the new library in ascending order
void ManageLibrary::insertDescriptorInOrderAscending(ManageLibrary& library, Descriptor* de
scriptor) {
    if (library.head == nullptr || library.head->getCost() > descriptor->getCost()) {
        // Insert at the beginning
        descriptor->setNextDescriptor(library.head);
        library.head = descriptor;
    } else {
        // Traverse the library to find the correct position
        Descriptor* current = library.head;
        while (current->getNextDescriptor() != nullptr && current->getNextDescriptor()->get
```

```
Cost() <= descriptor->getCost()) {
            current = current->getNextDescriptor();
        }
        descriptor->setNextDescriptor(current->getNextDescriptor());
        current->setNextDescriptor(descriptor);
    }
}

void ManageLibrary::setHead(Descriptor* head) {
    this->head = head;
}

Descriptor* ManageLibrary::getDescriptorsByMaxCost(double maxCost) {
    Descriptor* filteredHead = nullptr;
    Descriptor* filteredTail = nullptr;
    Descriptor* current = head;

    qDebug() << "Filtering descriptors by max cost:" << maxCost;

    while (current != nullptr) {
        if (current->getCost() <= maxCost) {
            Descriptor* newDescriptor = new Descriptor(*current); // Create a deep copy
            newDescriptor->setNextDescriptor(nullptr); // Ensure it's isolated

            if (filteredHead == nullptr) {
                filteredHead = newDescriptor;
                filteredTail = newDescriptor;
            } else {
                filteredTail->setNextDescriptor(newDescriptor);
                filteredTail = newDescriptor;
            }
        }
        current = current->getNextDescriptor();
    }
    filteredTail->setNextDescriptor(nullptr);

    qDebug() << "Filtering complete. Returning filtered descriptors.";
    return filteredHead;
}

QString ManageLibrary::getLibraryPath() const {
    return libraryPath;
}

void ManageLibrary::setLibraryPath(QString path) {
    libraryPath = path;
}

void ManageLibrary::saveLibraryToJson(QString libraryName) {
    // Create the root JSON array
    QJsonArray libraryArray;
    QString appPath = QCoreApplication::applicationDirPath();

    // Traverse the linked list of descriptors
    Descriptor* current = head;
    while (current != nullptr) {
        QJsonObject descriptorObject;

        // Add each property to the JSON object
        descriptorObject["Imagepath"] = current->getImage().getPath(); // Assuming Image cl
ass has getImagePath()
        descriptorObject["access"] = QString(current->getAccess());         // Convert char
 to QString
        descriptorObject["cost"] = current->getCost();
        descriptorObject["id"] = static_cast<int>(current->getIdDescriptor());
        descriptorObject["source"] = current->getSource();
        descriptorObject["title"] = current->getTitle();
```

```
        // Append the object to the array
        libraryArray.append(descriptorObject);

        // Move to the next descriptor
        current = current->getNextDescriptor();
    }

    // Create the root JSON object
    QJsonObject rootObject;
    rootObject["library"] = libraryArray;

    // Convert to JSON document
    QJsonDocument jsonDoc(rootObject);

    // Write the JSON document to a file
    QFile file(appPath + "/../config/json_config/libraries.json" + libraryName + ".json");
    if (file.open(QIODevice::WriteOnly)) {
        file.write(jsonDoc.toJson(QJsonDocument::Indented)); // Indented for readability
        file.close();
        qDebug() << "Library saved to library.json";
    } else {
        qWarning() << "Failed to save library: Unable to open file.";
    }
}

Descriptor* ManageLibrary::getDescriptorsBetweenMaxMinCost(double maxCost, double minCost)
{
    Descriptor* current = head;
    Descriptor* newHead = nullptr;
    Descriptor* newTail = nullptr;

    while (current != nullptr) {
        // Filter descriptors by cost
        if (current->getCost() >= minCost && current->getCost() <= maxCost) {
            // Create a copy of the current descriptor
            Descriptor* newDescriptor = new Descriptor(*current);
            newDescriptor->setNextDescriptor(nullptr); // Ensure the new descriptor has no
connections

            // Append the new descriptor to the new list
            if (newHead == nullptr) {
                newHead = newDescriptor;
                newTail = newDescriptor;
            } else {
                newTail->setNextDescriptor(newDescriptor);
                newTail = newDescriptor;
            }
        }
        current = current->getNextDescriptor();
    }

    return newHead;
}
```

```
#include "loginwindow.hpp"
#include "ui_loginwindow.h"
#include "user.hpp"
#include <QFile>
#include <QTextStream>
#include <QMessageBox>
#include <QScreen>
#include <QGuiApplication>
#include <QJsonDocument>
#include <QJsonObject>

LoginWindow::LoginWindow(User &user, QWidget *parent)
    : QDialog(parent), ui(new Ui::LoginWindow), user(user)
{
    ui->setupUi(this);                                      // Configurer la fenÃªtre
    this->setFixedSize(340 , 579);                          // Correction de la fenÃªtre de conn
exion (ne peut pas Ãªtre dÃ©placÃ©e)
    QPixmap pixmap(":/AppImages/Widget.png");               // Le chemin de l'image qui montre u
ne personne en PNG dans la fenÃªtre de connexion

    // Appliquer des styles aux Ã©lÃ©ments de l'interface
    ui->LoginButton->setStyleSheet("background-color: #01042e; color: white; padding: 10px
15px; border: none; border-radius: 5px; font-size: 10px;");
    ui->LoginInput->setStyleSheet("padding: 10px; border: 1px solid #ccc; border-radius: 5p
x;width: 300px; height: 80px;font-size: 10px;");
    ui->label->setStyleSheet("font-size: 22px; font-weight: bold; color: #01042e;");
    // DÃ©finir le titre et la taille de la fenÃªtre
    setWindowTitle("Login");
    ui->label_2->setStyleSheet(" color: #01042e;");    // Donner au texte "SIGN IN" une cou
leur bleu foncÃ©
    // DÃ©finir l'image de fond pour le label
    ui->backgroundImageLabel->setStyleSheet("background-image: url(:/AppImages/Back_g.jpg);
 background-position: center; background-repeat: no-repeat; background-size: cover;");
    ui->label_3->setPixmap(pixmap);                         // DÃ©finir l'image sur le QLabel
    ui->label_3->setScaledContents(true);                   // Faire en sorte que l'image soit r
edimensionnÃ©e avec le label
    ui->LoginInput->setEchoMode(QLineEdit::Password);  // Masquer le code
    ui->LoginInput->setEchoMode(QLineEdit::Password);

    ui->LoginButton->raise();                               // Placer ce bouton au-dessus de la
fenÃªtre
    ui->LoginInput->raise();                                // Placer ce bouton au-dessus de la
fenÃªtre
    ui->label->raise();                                     // Placer ce bouton au-dessus de la
fenÃªtre
}

LoginWindow::~LoginWindow()
{
    delete ui;
}

void LoginWindow::on_LoginButton_clicked()
{

    QString filePath = QCoreApplication::applicationDirPath() + "/auth.json"; // chemin ver
s le fichier auth.json oÃ¹ tous les codes sont stockÃ©s

    QFile file(filePath);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {

        QMessageBox::warning(nullptr, "Error", "Unable to open file for Authentfication!");
}               // Tester si le fichier peut Ãªtre ouvert

    QString codeInput = ui->LoginInput->text();
         //
    QByteArray data = file.readAll();
    QJsonDocument doc(QJsonDocument::fromJson(data));
```

```
    QJsonObject obj = doc.object();
    QJsonArray array = obj["users"].toArray();


    int i(0);
    bool isAuthenticated = false;  // Indicateur pour suivre l'Ã©tat de l'authentification

    for (int i = 0; i < array.size(); i++) {
        QJsonObject obj = array[i].toObject();

        if (obj["code"] == codeInput) {  // VÃ©rifier si codeInput correspond Ã  obj["code"
]
            isAuthenticated = true;  // Marquer comme authentifiÃ©

            if (obj["access"].toInt() == 0) {  // VÃ©rifier la valeur d'accÃ¨s
                qDebug() << "access  = " << obj["access"];
                user.access = false;
            } else {
                qDebug() << "access  = " << obj["access"];
                user.access = true;
            }

            accept();  // ProcÃ©der si authentifiÃ©
            break;     // Quitter la boucle car nous avons trouvÃ© une correspondance
        }
    }

    // Si aucune correspondance n'a Ã©tÃ© trouvÃ©e (isAuthenticated est toujours false)
    if (!isAuthenticated) {
        QMessageBox::warning(this, "Login Failed", "You are Not Authenticated");  // Affich
er un message si l'utilisateur n'est pas authentifiÃ©
    }

    // Effacer le texte dans le champ LoginInput aprÃ¨s traitement
    ui->LoginInput->clear();                            // Effacer le texte aprÃ¨s l'avoir ut
ilisÃ©
}
```

```
#include "mainwindow.h"
#include "loginwindow.hpp"
#include "user.hpp"
#include <QApplication>
#include <QLoggingCategory>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QLoggingCategory::setFilterRules("*.debug=false");

    User user; // Create a User object
    LoginWindow loginWindow(user); // Create the login window

    // Loop until user logs in or cancels login
    while (true) {
        // Show the login window and check authentication
        if (loginWindow.exec() == QDialog::Accepted) {
            // If login is successful, create the MainWindow with the user object
            MainWindow mainWindow(user);

            // Connect the logout signal from MainWindow to show the login window again
            QObject::connect(&mainWindow, &MainWindow::logoutRequested, [&]() {
                mainWindow.close();  // Close the MainWindow
                loginWindow.show();  // Show the LoginWindow again
            });

            // Show the main window
            mainWindow.show();

            // Run the application event loop for the MainWindow
            a.exec();

            // After logout, reset the login window (optional step to ensure no state persi
sts)
            loginWindow.hide(); // Hide the LoginWindow (to ensure it doesn't reappear in t
he background)
        } else {
            // If login fails or the user cancels, exit the application
            return 0;
        }
    }
}
```

```
#include "mainwindow.h"
#include "loginwindow.hpp"
#include "./ui_mainwindow.h"
#include <QFileDialog>
#include <QDebug>
#include <QLabel>
#include <QPixmap>
#include <QVBoxLayout>
#include <QFormLayout>
#include <QDialogButtonBox>
#include "librarymanagement.hpp"
#include "descriptor.hpp"
#include "add_new_descriptor.hpp"
#include <QJsonObject>
#include <QInputDialog>
#include <QMessageBox>
#include <QPushButton>
#include <QCoreApplication>
#include <QJsonDocument>



MainWindow::MainWindow(User user, QWidget *parent, ManageLibrary mainlibrary, ManageLibrary
 sublibrary)
    : QMainWindow(parent), ui(new Ui::Home), currentUser(user), mainlibrary(ManageLibrary(0
, nullptr, "")), sublibrary(ManageLibrary(0, nullptr, ""))
{
    // this->setFixedSize(1200, 800); // Width: 1200, Height: 800

    ui->setupUi(this);
    // this->mainlibrary = ManageLibrary(0, nullptr);

    descriptorDetails = new DescriptorDetails(this, currentUser.access);
    if (currentUser.access)
    {
        ui->menubar->setVisible(true);
    }
    else
    {
        ui->menubar->setVisible(false);
    }

    // Initialize the grid layout
    gridLayout = new QGridLayout();
    ui->librariesLayout->setLayout(gridLayout);
    loadLibrariesButtons();
    ui->LogoutButton->setVisible(true);

    ui->returnButton->setVisible(false);
    connect(ui->returnButton, &QPushButton::clicked, this, &MainWindow::on_returnButton_cli
cked);

    ui->saveSubListButton->setEnabled(false); // DÃ©sactive le bouton

    // Apply styles to the specified UI elements
    ui ->LogoutButton->setStyleSheet("QPushButton {"
                                          "background-color: rgb(153, 193, 241);"
                                          "color: white;"
                                          "border: none;"
                                          "border-radius: 5px;"
                                          "padding: 8px 12px;"
                                          "font-size: 14px;"
                                          "font-weight: bold;"
                                          "}"
                                          "QPushButton:hover {"
                                          "background-color: rgb(123, 163, 211);"
                                          "}"
                                          "QPushButton:pressed {"
```

```
                                            "background-color: #003f7f;"
                                            "padding-left: 12px;"
                                            "padding-top: 12px;"
                                            "}");
    ui->CreateSubListLabel->setStyleSheet("font-size: 18px; font-weight: bold; color: #333;
");
    ui->LoadLibraryLabel->setStyleSheet("font-size: 18px; font-weight: bold; color: #333;")
;
    ui->ImageIdSearchInput->setStyleSheet("padding: 10px; border: 1px solid #ccc; border-ra
dius: 5px;");
    ui->SearchButton->setStyleSheet("QPushButton {"
                                            "background-color: rgb(153, 193, 241);"
                                            "color: white;"
                                            "border: none;"
                                            "border-radius: 5px;"
                                            "padding: 8px 12px;"
                                            "font-size: 14px;"
                                            "font-weight: bold;"
                                            "}"
                                            "QPushButton:hover {"
                                            "background-color: rgb(123, 163, 211);"
                                            "}"
                                            "QPushButton:pressed {"
                                            "background-color: #003f7f;"
                                            "padding-left: 12px;"
                                            "padding-top: 12px;"
                                            "}");
    ui->comboBox_libraries->setStyleSheet("padding: 5px; border: 1px solid #ccc; border-rad
ius: 5px; background-color: #fff; color: #333;");
    ui->ClearFilterButton->setVisible(false);

    // Connect the combo box's currentIndexChanged signal to the slot
    connect(ui->comboBox_libraries, QOverload<int>::of(&QComboBox::currentIndexChanged), th
is, [this](int index)
            {
        QString libraryPath = ui->comboBox_libraries->itemData(index).toString();
        if (!libraryPath.isEmpty()) {
            LoadTheLibrary(libraryPath);
        } });
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::loadLibrariesButtons()
{

    // Clear existing items from the combo box
    ui->comboBox_libraries->clear();

    QString appPath = QCoreApplication::applicationDirPath();
    QString librariesFilePath = appPath + "/libraries.json";

    // Load libraries from the JSON file
    QJsonArray jsonArray = currentUser.loadLibraries(librariesFilePath);
    if (jsonArray.isEmpty())
    {
        QMessageBox::warning(this, "No Libraries Found",
                             "No libraries could be loaded from libraries.json. Please chec
k the file.");
        // qDebug() << "No libraries found in the file.";
        return; // Exit if there are no libraries
    }

    // qDebug() << "Loaded libraries: ";
```

```
    // Populate the combo box with library names and paths
    for (const QJsonValue &value : jsonArray)
    {
        if (!value.isObject())
            continue; // Skip non-object entries

        QJsonObject libraryObj = value.toObject();
        QString libraryName = libraryObj.value("name").toString();
        QString libraryPath = appPath + libraryObj.value("path").toString();

        if (libraryName.isEmpty() || libraryPath.isEmpty())
        {
            // qDebug() << "Invalid library entry: " << libraryObj;
            continue; // Skip invalid entries
        }

        ui->comboBox_libraries->addItem(libraryName, libraryPath);
    }

    if (ui->comboBox_libraries->count() > 0)
    {
        QString libraryPath = ui->comboBox_libraries->itemData(0).toString();
        if (!libraryPath.isEmpty())
        {
            LoadTheLibrary(libraryPath); // Load the first library by default
        }
    }
}

void MainWindow::on_actionLoad_a_Library_triggered()
{
    // Enter the library to import
    QString path = QFileDialog::getOpenFileName(this, "Open Library", "", "JSON files (*.js
on)");
    // print the path in the terminal
    // qDebug() << path;
    // show the library
    LoadTheLibrary(path);
}

void MainWindow::setCurrentLibraryPath(QString path){
    this->currentLibraryPath = path;
}

void MainWindow::LoadTheLibrary(QString path)
{
    this->setCurrentLibraryPath(path);

    // Reload the library from the file system
    ManageLibrary library = currentUser.loadLibrary(path);
    mainlibrary = library;

    // qDebug() << "Library Created";

    // If the library is empty
    if (library.getHead() == nullptr)
    {
        // qDebug() << "The library is empty";
        clearGridLayout();
        // ui->emptyLibrary->setText("The library is empty.");
        return;
    }

    // ui->emptyLibrary->setText("");
    double maxCost = mainlibrary.getMaxDescriptorCost();
    double minCost = mainlibrary.getMinDescriptorCost();

    // Clear the existing items in the grid layout
```

```
    clearGridLayout();

    // VÃ©rification du chemin de l'image
    QString imagePath = library.getLibraryPath(); // Utilise la mÃ©thode getLibraryPath pou
r obtenir le chemin
    if (QFile::exists(imagePath)) {
        // qDebug() << "Loading image: " << imagePath;
        // Continue to load image here
        populateGridLayout(library.getHead());
    } else {
        // Handle error if image cannot be loaded
        // qDebug() << "Error while loading the image: " << imagePath;
        QMessageBox::warning(this, "Image Error", "Unable to load the image. Please check t
he file path.");
        // Optionally, clear grid layout or set a default state
    }
}



void MainWindow::clearGridLayout()
{
    while (QLayoutItem *item = gridLayout->takeAt(0))
    {
        if (QWidget *widget = item->widget())
        {
            widget->deleteLater(); // Ensure proper deletion of the widget
        }
        delete item; // Delete the layout item
    }
}

void MainWindow::populateGridLayout(Descriptor *head)
{
    int row = 0;
    int col = 0;
    Descriptor *current = head;
    QString appPath = QCoreApplication::applicationDirPath();

    while (current != nullptr)
    {
        if(current->getAccess() == 'L' && !currentUser.access){
            current = current->getNextDescriptor();
            continue;
        }
        // Create a vertical layout for each cell
        QVBoxLayout *cellLayout = new QVBoxLayout();
        cellLayout->setContentsMargins(10, 10, 10, 10);
        cellLayout->setSpacing(10);

        // Create and add the image label
        QLabel *imageLabel = new QLabel();
        // qDebug() << "Loading Image : " << appPath + current->getImage().getPath();
        QPixmap pixmap(appPath + current->getImage().getPath());
        if (pixmap.isNull())
        {
            qWarning() << "Failed to load image: " << current->getImage().getPath();
        }
        imageLabel->setPixmap(pixmap.scaled(210, 210, Qt::KeepAspectRatio));
        imageLabel->setStyleSheet("border: 1px solid #ccc; padding: 5px;");
        cellLayout->addWidget(imageLabel);

        // // Create and add the information label
        QLabel *infoLabel = new QLabel();
        QString infoText = QString("ID: %1").arg(current->getIdDescriptor());
        infoLabel->setText(infoText);
        infoLabel->setStyleSheet("background-color: #f9f9f9; padding: 10px; border-radius:
5px;");
```

```
        infoLabel->setFixedSize(240, 33);

        cellLayout->addWidget(infoLabel);

        // Create an info button
            QPushButton *infoButton = new QPushButton("Show/Hide Info", this);
            infoButton->setStyleSheet(  "QPushButton {"
                                        "background-color: rgb(153, 193, 241);"
                                        "color: white;"
                                        "border: none;"
                                        "border-radius: 5px;"
                                        "padding: 8px 12px;"
                                        "font-size: 14px;"
                                        "font-weight: bold;"
                                        "}"
                                        "QPushButton:hover {"
                                        "background-color: rgb(123, 163, 211);"
                                        "}"
                                        "QPushButton:pressed {"
                                        "background-color: #003f7f;"
                                        "padding-left: 12px;"
                                        "padding-top: 12px;"
                                        "}");
            cellLayout->addWidget(infoButton);

        // If the user has access, create a delete button
        if (getCurrentUser().access)
        {
            QPushButton *deleteButton = new QPushButton("Delete", this);
            deleteButton->setStyleSheet(  "QPushButton {"
                                        "background-color: rgb(153, 193, 241);"
                                        "color: white;"
                                        "border: none;"
                                        "border-radius: 5px;"
                                        "padding: 8px 12px;"
                                        "font-size: 14px;"
                                        "font-weight: bold;"
                                        "}"
                                        "QPushButton:hover {"
                                        "background-color: rgb(123, 163, 211);"
                                        "}"
                                        "QPushButton:pressed {"
                                        "background-color: #003f7f;"
                                        "padding-left: 12px;"
                                        "padding-top: 12px;"
                                        "}");
            cellLayout->addWidget(deleteButton);

            // Use a lambda to delete the descriptor
            connect(deleteButton, &QPushButton::clicked, this, [this, descriptor = current]
()
                    {
                        mainlibrary.deleteDescriptor(descriptor);
                        ShowTheLibrary(mainlibrary); // Reload the library after deletion
                    });

            QPushButton *editButton = new QPushButton("Edit", this);
            editButton->setStyleSheet(  "QPushButton {"
                                        "background-color: rgb(153, 193, 241);"
                                        "color: white;"
                                        "border: none;"
                                        "border-radius: 5px;"
                                        "padding: 8px 12px;"
                                        "font-size: 14px;"
                                        "font-weight: bold;"
                                        "}"
                                        "QPushButton:hover {"
                                        "background-color: rgb(123, 163, 211);"
```

```
                                            "}"
                                            "QPushButton:pressed {"
                                            "background-color: #003f7f;"
                                            "padding-left: 12px;"
                                            "padding-top: 12px;"
                                            "}");
            cellLayout->addWidget(editButton);

            // Connect the Edit button to display a QMessageBox
            connect(editButton, &QPushButton::clicked, this, [this, current]()
                {

                    unsigned int originalId = current->getIdDescriptor();

                    QDialog dialog(this);
                    dialog.setWindowTitle("Edit Image Info");
                    dialog.setModal(true);

                    QLineEdit *idEdit = new QLineEdit(QString::number(current->getIdDescrip
tor()), &dialog);
                    QLineEdit *titleEdit = new QLineEdit(current->getTitle(), &dialog);
                    QLineEdit *sourceEdit = new QLineEdit(current->getSource(), &dialog);
                    QLineEdit *costEdit = new QLineEdit(QString::number(current->getCost())
, &dialog);

                    QComboBox *accessCombo = new QComboBox(&dialog);
                    accessCombo->addItem("L");
                    accessCombo->addItem("O");
                    accessCombo->setCurrentText(QString(current->getAccess()));
                    // CrÃ©er un layout pour organiser les champs
                    QFormLayout *formLayout = new QFormLayout();
                    formLayout->addRow("ID:", idEdit);
                    formLayout->addRow("Title:", titleEdit);
                    formLayout->addRow("Source:", sourceEdit);
                    formLayout->addRow("Cost:", costEdit);
                    formLayout->addRow("Access:", accessCombo);


                    // Ajouter les boutons
                    QDialogButtonBox *buttonBox = new QDialogButtonBox(QDialogButtonBox::Sa
ve | QDialogButtonBox::Cancel, &dialog);

                    // Connecter les boutons
                    connect(buttonBox, &QDialogButtonBox::accepted, &dialog, &QDialog::acce
pt);
                    connect(buttonBox, &QDialogButtonBox::rejected, &dialog, &QDialog::reje
ct);

                    // Organiser le tout dans un layout principal
                    QVBoxLayout *mainLayout = new QVBoxLayout(&dialog);
                    mainLayout->addLayout(formLayout);
                    mainLayout->addWidget(buttonBox);

                    // Afficher la boÃ®te de dialogue
                    if (dialog.exec() == QDialog::Accepted) {
                        // Mettre Ã  jour les informations
                        current->setIdDescriptor(idEdit->text().toInt());

                        current->setTitle(titleEdit->text());
                        current->setSource(sourceEdit->text());
                        current->setCost(costEdit->text().toDouble());
                        current->setAccess(accessCombo->currentText().toStdString()[0]); //
 RÃ©cupÃ©rer la valeur sÃ©lectionnÃ©e

                        SaveChanges_clicked(current, originalId);

                        ShowTheLibrary(mainlibrary); // RafraÃ®chir l'affichage
                    }
```

```
                });
            }


                bool *isInfoVisible = new bool(false); // Initial state: info hidden

                // Connect the info button
                connect(infoButton, &QPushButton::clicked, this, [infoLabel, current, isInf
oVisible]()
                    {
                        if (*isInfoVisible) {
                        // Hide additional information
                        infoLabel->setText(QString("ID: %1").arg(current->getIdDescriptor()
));
                        infoLabel->setFixedSize(240, 33); // Revenir Ã  la taille initiale
                        *isInfoVisible = false;

                        } else {
                            // Show additional information
                            QString additionalInfo = QString("\nCost: %1\nTitle: %2\nSource
: %3\nAccess: %4")
                                                .arg(current->getCost())         // Cost
                                                .arg(current->getTitle())        // Titre
                                                .arg(current->getSource())       // Source
                                                .arg(current->getAccess());        // Access
                            infoLabel->setText(infoLabel->text() + additionalInfo);

                            infoLabel->setFixedSize(QWIDGETSIZE_MAX, QWIDGETSIZE_MAX);
                            infoLabel->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::P
referred); // Autoriser l'expansion
                            *isInfoVisible = true;
                }
        });

        // Create a widget to hold the cell layout and add it to the grid layout
        QWidget *cellWidget = new QWidget();
        cellWidget->setLayout(cellLayout);
        cellWidget->setFixedSize(250, 350);
        cellWidget->setStyleSheet("background-color: #ffffff; border: 1px solid #ddd; borde
r-radius: 10px; padding: 10px;");
        gridLayout->addWidget(cellWidget, row, col);

        // Store the connection between the widget and the descriptor
        widgetDescriptorMap[cellWidget] = current;

        // Install an event filter for the widget
        cellWidget->installEventFilter(this);

        // Update row and column for grid layout
        col++;
        if (col >= 3)
        {
            col = 0;
            row++;
        }

        current = current->getNextDescriptor();
    }
}

void MainWindow::cleanUpDescriptors(Descriptor *head)
{
    Descriptor *current = head;
    while (current != nullptr)
    {
        Descriptor *temp = current;
        current = current->getNextDescriptor();
        delete temp;
```

```
    }
}

void MainWindow::ShowTheLibrary(ManageLibrary library)
{
    // qDebug() << "To show the library";

    // If the library is empty
    if (library.getHead() == nullptr)
    {
        // qDebug() << "The library is empty";
        clearGridLayout();
        QMessageBox::warning(this, "Warning", "The library is empty.");
        return;
    }

    // Clear the existing items in the grid layout
    clearGridLayout();

    // Populate the grid layout with images and their information
    populateGridLayout(library.getHead());
}

User MainWindow::getCurrentUser()
{
    return this->currentUser;
};
bool MainWindow::eventFilter(QObject *obj, QEvent *event)
{
    if (event->type() == QEvent::MouseButtonPress)
    {
        QWidget *widget = qobject_cast<QWidget *>(obj);
        if (widget && widgetDescriptorMap.contains(widget))
        {
            Descriptor *descriptor = widgetDescriptorMap[widget];
            // qDebug() << "library Path in details" << this->currentLibraryPath;
            descriptorDetails->setLibraryPath(this->currentLibraryPath);
            descriptorDetails->setDescriptor(descriptor);
            descriptorDetails->show();
            return true;
        }
    }
    return QMainWindow::eventFilter(obj, event);
}

void MainWindow::refreshLibrary()
{
}

void MainWindow::on_CreateNewLibrary_triggered()
{
    bool ok;
    QString libraryName = QInputDialog::getText(this, tr("Create New Library"),
                                                tr("Library Name:"), QLineEdit::Normal, "",
 &ok);

    if (ok && !libraryName.isEmpty())
    {
        currentUser.createLibrary(libraryName);

        // Block signals temporarily to avoid triggering `currentIndexChanged`
        ui->comboBox_libraries->blockSignals(true);

        ui->comboBox_libraries->clear();
        QString appPath = QCoreApplication::applicationDirPath();
        QString librariesFilePath = appPath + "/libraries.json";
        QJsonArray jsonArray = currentUser.loadLibraries(librariesFilePath);
```

```
        for (const QJsonValue &value : jsonArray)
        {
            QJsonObject libraryObj = value.toObject();
            QString libraryName = libraryObj["name"].toString();
            QString libraryPath = libraryObj["path"].toString();

            // Validate and log paths
            QString fullPath = appPath + libraryPath;
            // qDebug() << "Adding to combo box: " << libraryName << " -> " << fullPath;

            ui->comboBox_libraries->addItem(libraryName, fullPath); // Store the absolute p
ath
        }

        // Unblock signals after populating
        ui->comboBox_libraries->blockSignals(false);
    }
    else
    {
        QMessageBox::warning(this, tr("Invalid Input"), tr("Library name cannot be empty.")
);
    }
}

void MainWindow::on_actionAdd_New_Descriptor_triggered()
{
    // qDebug() << "In MainWindow::on_add_new_description_clicked():";
    // qDebug() << "----------------------------------------";
    // qDebug() << MainWindow::getCurrentLibraryId();
    Add_New_Descriptor addDescriptorDialog(mainlibrary.getLibraryPath(), this);
    addDescriptorDialog.exec();
    // refresh the ui to show the new descriptor
    LoadTheLibrary(mainlibrary.getLibraryPath());
}

void MainWindow::on_actionDelete_a_library_triggered()
{
    // Load the libraries from the JSON file
    QString appPath = QCoreApplication::applicationDirPath();

    QString librariesFilePath = appPath + "/libraries.json";
    QJsonArray jsonArray = currentUser.loadLibraries(librariesFilePath);

    // Prepare a list of library names
    QStringList libraryNames;
    QMap<QString, QString> libraryPaths; // Map to store library names and their paths

    for (const QJsonValue &value : jsonArray)
    {
        QJsonObject libraryObj = value.toObject();
        QString libraryName = libraryObj["name"].toString();
        QString libraryPath = libraryObj["path"].toString();
        libraryNames.append(libraryName);        // Add library name to the list
        libraryPaths[libraryName] = libraryPath; // Store the corresponding library path
    }

    // Show a dialog to let the user choose which library to delete
    bool ok;
    QString libraryNameToDelete = QInputDialog::getItem(this, tr("Delete a Library"),
                                                        tr("Select Library to Delete:"),
                                                        libraryNames, 0, false, &ok);

    // If the user selected a library and clicked OK
    if (ok && !libraryNameToDelete.isEmpty())
    {
        // Get the path of the selected library
        QString libraryPath = libraryPaths[libraryNameToDelete];
```

```
        // Call deleteLibrary to delete the library
        currentUser.deleteLibrary(libraryNameToDelete);

        // Refresh the UI after deletion
        // qDebug() << "The library has been deleted:" << libraryNameToDelete;
        loadLibrariesButtons(); // Reload the buttons

        // Optionally, refresh or reload other parts of the UI
        // refreshLibrary(); // Uncomment if needed
    }
    else
    {
        // qDebug() << "No library was selected for deletion.";
    }
}

void MainWindow::on_SearchButton_clicked()
{
    QString ImageId = ui->ImageIdSearchInput->text();
    bool imageFound = false;
    Descriptor *current = mainlibrary.getHead();
    QString appPath = QCoreApplication::applicationDirPath();

    while (current != nullptr)
    {
        if(current->getAccess() == 'L' && !currentUser.access){
            current = current->getNextDescriptor();
            continue;
        }

        // check if the current descriptor id is equal to the id entered by the user
        if (current->getIdDescriptor() == ImageId.toInt())
        {
            imageFound = true;

            // Show the return button
            ui->returnButton->setVisible(true);
            // clear the grid layout
            QLayoutItem *item;
            while ((item = gridLayout->takeAt(0)) != nullptr)
            {
                delete item->widget();
                delete item;
            }

            // Create a vertical layout for each cell
            QVBoxLayout *cellLayout = new QVBoxLayout();
            cellLayout->setContentsMargins(10, 10, 10, 10);
            cellLayout->setSpacing(10);

            // Create and add the image label
            QLabel *imageLabel = new QLabel();
            QPixmap pixmap(appPath + current->getImage().getPath());
            imageLabel->setPixmap(pixmap.scaled(210, 210, Qt::KeepAspectRatio)); // Adjust
the size as needed
            imageLabel->setStyleSheet("border: 1px solid #ccc; padding: 5px;");
            cellLayout->addWidget(imageLabel);

            // // Create and add the information label
            QLabel *infoLabel = new QLabel();
            QString infoText = QString("ID: %1").arg(current->getIdDescriptor());
            infoLabel->setText(infoText);
            infoLabel->setStyleSheet("background-color: #f9f9f9; padding: 10px; border-radi
us: 5px;");
            infoLabel->setFixedSize(240, 33);
            cellLayout->addWidget(infoLabel);

            // Create an info button
```

```
            QPushButton *infoButton = new QPushButton("Show/Hide Info", this);
            infoButton->setStyleSheet(  "QPushButton {"
                                        "background-color: rgb(153, 193, 241);"
                                        "color: white;"
                                        "border: none;"
                                        "border-radius: 5px;"
                                        "padding: 8px 12px;"
                                        "font-size: 14px;"
                                        "font-weight: bold;"
                                        "}"
                                        "QPushButton:hover {"
                                        "background-color: rgb(123, 163, 211);"
                                        "}"
                                        "QPushButton:pressed {"
                                        "background-color: #003f7f;"
                                        "padding-left: 12px;"
                                        "padding-top: 12px;"
                                        "}");
            cellLayout->addWidget(infoButton);

            if (getCurrentUser().access)
            {
                QPushButton *deleteButton = new QPushButton("Delete", this);
                deleteButton->setStyleSheet(  "QPushButton {"
                                        "background-color: rgb(153, 193, 241);"
                                        "color: white;"
                                        "border: none;"
                                        "border-radius: 5px;"
                                        "padding: 8px 12px;"
                                        "font-size: 14px;"
                                        "font-weight: bold;"
                                        "}"
                                        "QPushButton:hover {"
                                        "background-color: rgb(123, 163, 211);"
                                        "}"
                                        "QPushButton:pressed {"
                                        "background-color: #003f7f;"
                                        "padding-left: 12px;"
                                        "padding-top: 12px;"
                                        "}");
                cellLayout->addWidget(deleteButton);

                // Use a lambda to delete the descriptor
                connect(deleteButton, &QPushButton::clicked, this, [this, descriptor = curr
ent]()
                    {
                        mainlibrary.deleteDescriptor(descriptor);
                        ShowTheLibrary(mainlibrary); // Reload the library after deletion
                    });
                QPushButton *editButton = new QPushButton("Edit", this);
                editButton->setStyleSheet(  "QPushButton {"
                                        "background-color: rgb(153, 193, 241);"
                                        "color: white;"
                                        "border: none;"
                                        "border-radius: 5px;"
                                        "padding: 8px 12px;"
                                        "font-size: 14px;"
                                        "font-weight: bold;"
                                        "}"
                                        "QPushButton:hover {"
                                        "background-color: rgb(123, 163, 211);"
                                        "}"
                                        "QPushButton:pressed {"
                                        "background-color: #003f7f;"
                                        "padding-left: 12px;"
                                        "padding-top: 12px;"
                                        "}");
            cellLayout->addWidget(editButton);
```

```
            connect(editButton, &QPushButton::clicked, this, [this, current]()
                {

                    unsigned int originalId = current->getIdDescriptor();

                    QDialog dialog(this);
                    dialog.setWindowTitle("Edit Image Info");
                    dialog.setModal(true);

                    QLineEdit *idEdit = new QLineEdit(QString::number(current->getIdDescrip
tor()), &dialog);
                    QLineEdit *titleEdit = new QLineEdit(current->getTitle(), &dialog);
                    QLineEdit *sourceEdit = new QLineEdit(current->getSource(), &dialog);
                    QLineEdit *costEdit = new QLineEdit(QString::number(current->getCost())
, &dialog);

                    QComboBox *accessCombo = new QComboBox(&dialog);
                    accessCombo->addItem("L");
                    accessCombo->addItem("O");
                    accessCombo->setCurrentText(QString(current->getAccess()));
                    // CrÃ©er un layout pour organiser les champs
                    QFormLayout *formLayout = new QFormLayout();
                    formLayout->addRow("ID:", idEdit);
                    formLayout->addRow("Title:", titleEdit);
                    formLayout->addRow("Source:", sourceEdit);
                    formLayout->addRow("Cost:", costEdit);
                    formLayout->addRow("Access:", accessCombo);


                    // Ajouter les boutons
                    QDialogButtonBox *buttonBox = new QDialogButtonBox(QDialogButtonBox::Sa
ve | QDialogButtonBox::Cancel, &dialog);

                    // Connecter les boutons
                    connect(buttonBox, &QDialogButtonBox::accepted, &dialog, &QDialog::acce
pt);
                    connect(buttonBox, &QDialogButtonBox::rejected, &dialog, &QDialog::reje
ct);

                    // Organiser le tout dans un layout principal
                    QVBoxLayout *mainLayout = new QVBoxLayout(&dialog);
                    mainLayout->addLayout(formLayout);
                    mainLayout->addWidget(buttonBox);

                    // Afficher la boÃ®te de dialogue
                    if (dialog.exec() == QDialog::Accepted) {
                        // Mettre Ã  jour les informations
                        current->setIdDescriptor(idEdit->text().toInt());

                        current->setTitle(titleEdit->text());
                        current->setSource(sourceEdit->text());
                        current->setCost(costEdit->text().toDouble());
                        current->setAccess(accessCombo->currentText().toStdString()[0]); //
 RÃ©cupÃ©rer la valeur sÃ©lectionnÃ©e

                        SaveChanges_clicked(current, originalId);

                        ShowTheLibrary(mainlibrary); // RafraÃ®chir l'affichage
                    }
                });
            }

            bool *isInfoVisible = new bool(false);

                // Connect the info button
                connect(infoButton, &QPushButton::clicked, this, [infoLabel, current, isInf
oVisible]()
```

```
                    {
                        if (*isInfoVisible) {
                        // Hide additional information
                        infoLabel->setText(QString("ID: %1").arg(current->getIdDescriptor()
));
                        infoLabel->setFixedSize(240, 33); // Revenir Ã  la taille initiale
                        *isInfoVisible = false;

                        } else {
                            // Show additional information
                            QString additionalInfo = QString("\nCost: %1\nTitle: %2\nSource
: %3\nAccess: %4")
                                                    .arg(current->getCost())         // Cost
                                                    .arg(current->getTitle())        // Titre
                                                    .arg(current->getSource())       // Source
                                                    .arg(current->getAccess());        // Access
                            infoLabel->setText(infoLabel->text() + additionalInfo);

                            infoLabel->setFixedSize(QWIDGETSIZE_MAX, QWIDGETSIZE_MAX);
                            infoLabel->setSizePolicy(QSizePolicy::Preferred, QSizePolicy::P
referred); // Autoriser l'expansion
                            *isInfoVisible = true;
                    }
        });

            // Create a widget to hold the cell layout and add it to the grid layout
            QWidget *cellWidget = new QWidget();
            cellWidget->setLayout(cellLayout);
            cellWidget->setFixedSize(250, 350); // Set fixed size for each descriptor
            cellWidget->setStyleSheet("background-color: #ffffff; border: 1px solid #ddd; b
order-radius: 10px; padding: 10px;");
            gridLayout->addWidget(cellWidget, 0, 0);

            // Store the connection between the widget and the descriptor
            widgetDescriptorMap[cellWidget] = current;

            // Connect the click event to the slot
            cellWidget->installEventFilter(this);
        }
        current = current->getNextDescriptor();
    }

    if (!imageFound)
    {
        QMessageBox::warning(this, "Error", "No image found with this ID.");
    }
}
void MainWindow::on_returnButton_clicked()
{
    ShowTheLibrary(mainlibrary);
    // Show the return button
    ui->returnButton->setVisible(false);
}



void MainWindow::on_DescendingButton_clicked()
{
    // order the descriptors by cost in descending order
    ManageLibrary orderedLibrary = mainlibrary.orderDescriptorsByCostDescending();
    mainlibrary = orderedLibrary;
    // refresh the library
    ShowTheLibrary(mainlibrary);
}

void MainWindow::on_AscendingButton_clicked()
{
    ManageLibrary orderedLibrary = mainlibrary.orderDescriptorsByCostAscending();
```

```
    mainlibrary = orderedLibrary;
    // refresh the library
    ShowTheLibrary(mainlibrary);
}

void MainWindow::on_saveSubListButton_clicked()
{
    // show a box to enter the name of the library
    bool ok;
    QString libraryName = QInputDialog::getText(this, tr("Save Sublibrary"),
                                                tr("Sublibrary Name:"), QLineEdit::Normal,
"", &ok);

    // Save the sublibrary to a JSON file

    sublibrary.saveLibraryToJson(libraryName);
    // save the library name and path in libraries.json file
    QString appPath = QCoreApplication::applicationDirPath();
    QString librariesFilePath = appPath + "/libraries.json";

    QJsonObject library;
    library["name"] = libraryName;
    library["path"] = "/Libraries/" + libraryName + ".json";
    // add to the libraries.json file
    QFile file(librariesFilePath);

    if (!file.open(QIODevice::ReadOnly))
    {
        QMessageBox::information(this,"Error:"," Could not open file");
    }

    QByteArray data = file.readAll();
    // add library to data
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["libraries"].toArray();
    array.append(library);
    obj["libraries"] = array;
    file.close();
    // write the updated data to the file
    if (!file.open(QIODevice::WriteOnly))
    {
        // qDebug() << "Error: Could not open file";
        exit(1);
    }
    file.write(QJsonDocument(obj).toJson());
    file.close();

    QMessageBox::information(this, "Success", "Sublibrary saved successfully.");
    loadLibrariesButtons();
}
void MainWindow::on_ClearFilterButton_clicked()
{
    // RÃ©initialiser les champs de saisie
    ui->MaxInput->clear();
    ui->MinInput->clear();
    ui->MaxInput_Only->clear();
    ui->MinInput_Only->clear();
    ui->Gratuit_checkBox->setChecked(false); // DÃ©cocher la case "Gratuit"

    // RÃ©afficher la liste complÃ¨te
    sublibrary.setHead(mainlibrary.getHead());
    ShowTheLibrary(sublibrary);

    // Cacher le bouton "Clear Filter" aprÃ¨s rÃ©initialisation
    ui->ClearFilterButton->setVisible(false);
}
```

```
void MainWindow::on_SubListButton_MaxMin_clicked()
{
    if (ui->MaxInput->text().isEmpty() || ui->MinInput->text().isEmpty())
    {
        QMessageBox::warning(this, "Error", "Please enter valid values for the cost.");
        return;
    }

    double maxCost = ui->MaxInput->text().toDouble();
    double minCost = ui->MinInput->text().toDouble();

    if (maxCost < minCost)
    {
        QMessageBox::warning(this, "Error", "The maximum cost must be greater than or equal
 to the minimum cost.");
        return;
    }

    Descriptor *newHead = mainlibrary.getDescriptorsBetweenMaxMinCost(maxCost, minCost);
    sublibrary.setHead(newHead);
    ShowTheLibrary(sublibrary);
    ui->ClearFilterButton->setVisible(true);


}

void MainWindow::on_SubListButton_Max_clicked()
{
    if (ui->MaxInput_Only->text().isEmpty())
    {
        QMessageBox::warning(this, "Error", "Please enter a value for the maximum cost.");
        return;
    }

    double maxCost = ui->MaxInput_Only->text().toDouble();

    if (maxCost < 0)
    {
        QMessageBox::warning(this, "Error", "The maximum cost must be positive.");
        return;
    }

    Descriptor *newHead = mainlibrary.getDescriptorsBetweenMaxMinCost(maxCost, 0);
    sublibrary.setHead(newHead);
    ShowTheLibrary(sublibrary);
    ui->ClearFilterButton->setVisible(true);


}

void MainWindow::on_SubListButton_Min_clicked()
{
    if (ui->MinInput_Only->text().isEmpty())
    {
        QMessageBox::warning(this, "Error", "Please enter a value for the minimum cost.");
        return;
    }

    double minCost = ui->MinInput_Only->text().toDouble();

    if (minCost < 0)
    {
        QMessageBox::warning(this, "Error", "The minimum cost must be positive.");
        return;
    }

    Descriptor *newHead = mainlibrary.getDescriptorsBetweenMaxMinCost(INFINITY, minCost);
    sublibrary.setHead(newHead);
```

```
    ShowTheLibrary(sublibrary);
    ui->ClearFilterButton->setVisible(true);


}

void MainWindow::on_SubListButton_Gratuit_clicked()
{
    bool gratuit = ui->Gratuit_checkBox->isChecked();

    Descriptor *newHead = nullptr;

    if (gratuit)
    {
        // Si la case est cochÃ©e, on filtre uniquement les Ã©lÃ©ments gratuits (cost = 0)
        newHead = mainlibrary.getDescriptorsBetweenMaxMinCost(0, 0);
    }
    else
    {
        // Si la case est dÃ©cochÃ©e, on filtre pour NE PAS afficher les gratuits (cost > 0
)
        newHead = mainlibrary.getDescriptorsBetweenMaxMinCost(INFINITY, 0.01);
    }

    sublibrary.setHead(newHead);
    ShowTheLibrary(sublibrary);
    ui->ClearFilterButton->setVisible(true);
}

void MainWindow::on_LogoutButton_clicked()
{
    emit logoutRequested();

}

void MainWindow::SaveChanges_clicked(Descriptor *currentDescriptor, unsigned int originalId
)
{
    QString libraryPath = this->currentLibraryPath;

    // RÃ©cupÃ©rer les nouvelles informations
    QJsonObject curObj = currentDescriptor->toJson();

    // Charger la bibliothÃ¨que
    QFile file(libraryPath);
    if (!file.open(QIODevice::ReadOnly)) {
        // qDebug() << "Error: Could not open file";
        return;
    }

    // Lire le fichier JSON existant
    QByteArray data = file.readAll();
    file.close();
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["library"].toArray();
    QJsonArray newArray;

    for (int i = 0; i < array.size(); i++) {
        QJsonObject obj = array[i].toObject();

        // VÃ©rifier si l'ID correspond Ã  l'ID d'origine
        if (obj["id"].toInt() == (int)originalId) {
            newArray.append(curObj); // Remplacer l'entrÃ©e
        } else {
            newArray.append(obj); // Garder les autres entrÃ©es inchangÃ©es
        }
    }
```

```
    obj["library"] = newArray;

    // Sauvegarder les modifications
    if (!file.open(QIODevice::WriteOnly)) {
        // qDebug() << "Error: Could not open file";
        return;
    }
    file.write(QJsonDocument(obj).toJson());
    file.close();

    // qDebug() << "Changes saved to the library file for ID:" << originalId;
}
```

```
#include "user.hpp"
#include "librarymanagement.hpp"
#include <QString>
#include <QJsonDocument>
#include <QJsonObject>
#include <QJsonArray>
#include <QFile>
#include <QIODevice>
#include <QDebug>
#include <QPushButton>
#include <QCoreApplication>

User::User(bool access):access(access) {}

ManageLibrary User::loadLibrary(const QString& path) const {
    // Load the file that contains the information of the library and create the ManageLibr
ary object
    // and display the library
    QFile file(path);
    if (!file.open(QIODevice::ReadOnly)) {
        qDebug() << "Error: Could not open file";
        exit(1);
    }
    qDebug() << "In load library : " << path;

    QByteArray data = file.readAll();
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["library"].toArray();

    if (array.isEmpty()) {
        qDebug() << "The library is empty.";
        file.close();
        ManageLibrary library(1, nullptr,path);

        return library; // Return an empty ManageLibrary object
    }


    Descriptor* head = nullptr;
    Descriptor* current = nullptr;

    for (int i = 0; i < array.size(); i++) {
        QJsonObject obj = array[i].toObject();
        qDebug() << "ID: " << obj["id"].toInt();
        // qDebug() << "Library ID: " << obj["libraryID"].toInt();
        qDebug() << "Cost: " << obj["cost"].toDouble();
        qDebug() << "Title: " << obj["title"].toString();
        qDebug() << "Source: " << obj["source"].toString();
        qDebug() << "Access: " << obj["access"].toString();
        qDebug() << "Imagepath: " << obj["Imagepath"].toString();


        Descriptor* newDescriptor = new Descriptor(
            obj["id"].toInt(),
            obj["cost"].toDouble(),
            obj["title"].toString(),
            obj["source"].toString(),
            obj["access"].toString().toStdString().c_str()[0],
            Image(obj["Imagepath"].toString())
            );
            if (head == nullptr) {
            head = newDescriptor;
            current = head;
        } else {
            current->setNextDescriptor(newDescriptor);
            current = newDescriptor;
        }
```

```
    }

    file.close();

    ManageLibrary library(1, head,path);
    qDebug() << "Displaying Library second time";
    library.display();
    return library;
}




QJsonArray User::loadLibraries(const QString& librariesFilePath) {
    qDebug() << "in loadLibraries function";
    // Load the file that contains the libraries path
    qDebug() << "Loading libraries from: " << librariesFilePath;
    QFile file(librariesFilePath);

    if (!file.exists()) {
        qDebug() << "Error: File does not exist";
        exit(1);
    }

    if (!file.open(QIODevice::ReadOnly)) {
        qDebug() << "Error: Could not open file";
        qDebug() << "Error details: " << file.errorString();
        exit(1);
    }

    QByteArray data = file.readAll();
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["libraries"].toArray();

    return array;
}


// create a new library method
void User::createLibrary(QString libraryName) {
  // create a new library json file and save it in the libraries folder
  // the file will contain an empty array
    QString appPath = QCoreApplication::applicationDirPath();
    qDebug() << "Creating library: " << libraryName;
    qDebug() << appPath;

  QJsonObject library;
  library["name"] = libraryName;
  library["path"] = "/Libraries/" + libraryName + ".json";

  //add to the libraries.json file
  QFile file(appPath + "/libraries.json");

  if (!file.open(QIODevice::ReadOnly)) {
      qDebug() << "Error: Could not open file";
      exit(1);
  }


  QByteArray data = file.readAll();
  // add library to data
  QJsonDocument doc(QJsonDocument::fromJson(data));
  QJsonObject obj = doc.object();
  QJsonArray array = obj["libraries"].toArray();
  array.append(library);
  obj["libraries"] = array;
  file.close();
```

```
  // write the updated data to the file
  if (!file.open(QIODevice::WriteOnly)) {
      qDebug() << "Error: Could not open file";
      exit(1);
  }
  file.write(QJsonDocument(obj).toJson());
  file.close();
  // create the library file
  qDebug() << "Creating the library file";
  qDebug() << appPath + library["path"].toString();

  QFile libraryFile(appPath + library["path"].toString());
  if (!libraryFile.open(QIODevice::WriteOnly)) {
      qDebug() << "Error: Could not open file";
      exit(1);
  }

  libraryFile.write(QJsonDocument(QJsonObject{{"library", QJsonArray()}}).toJson());
  libraryFile.close();


}
// create the delete library method that delete the library file and the librray from the l
ibraries.json file
void User::deleteLibrary(QString libraryToDelete){
    qDebug() << "Deleting library: " << libraryToDelete;
    QString appPath = QCoreApplication::applicationDirPath();

    QString libraryPath = appPath + "/Libraries/" + libraryToDelete + ".json";
    // delete the file
    qDebug() << "Deleting the library file at : " << libraryPath;
    QFile file(libraryPath);
    if (!file.open(QIODevice::ReadOnly)) {
        qDebug() << "Error: Could not open file";
        exit(1);
    }
    file.remove();
    file.close();
    // delete the name and path of the library from the libraries.json file
    QFile librariesFile(appPath + "/libraries.json");
    if (!librariesFile.open(QIODevice::ReadOnly)) {
        qDebug() << "Error: Could not open file";
        exit(1);
    }
    QByteArray data = librariesFile.readAll();
    QJsonDocument doc(QJsonDocument::fromJson(data));
    QJsonObject obj = doc.object();
    QJsonArray array = obj["libraries"].toArray();
    QJsonArray newArray;
    for (int i = 0; i < array.size(); i++) {
        QJsonObject libraryObj = array[i].toObject();
        if (libraryObj["name"].toString() != libraryToDelete) {
            newArray.append(libraryObj);
        }
    }
    obj["libraries"] = newArray;
    librariesFile.close();
    if (!librariesFile.open(QIODevice::WriteOnly)) {
        qDebug() << "Error: Could not open file";
        exit(1);
    }
    librariesFile.write(QJsonDocument(obj).toJson());
    librariesFile.close();

}
```

```
#ifndef ADD_NEW_DESCRIPTOR_HPP
#define ADD_NEW_DESCRIPTOR_HPP

#include <QDialog>

namespace Ui {
class Add_New_Descriptor;
}

class Add_New_Descriptor : public QDialog
{
    Q_OBJECT

public:
    explicit Add_New_Descriptor(QString Librarypath,QWidget *parent = nullptr);
    ~Add_New_Descriptor();
    void setLibraryPath(QString Librarypath);

private slots:
    void on_loadImageButton_clicked();

    void on_save_the_descriptor_clicked();

private:
    Ui::Add_New_Descriptor *ui;
    QString Librarypath;

};

#endif // ADD_NEW_DESCRIPTOR_HPP
```

```
#ifndef CLICKABLELABEL_HPP
#define CLICKABLELABEL_HPP

#include <QLabel>
#include <QWidget>
#include <QMouseEvent>

class ClickableLabel : public QLabel {
    Q_OBJECT

public:
    explicit ClickableLabel(QWidget *parent = nullptr) : QLabel(parent) {}

signals:
    void clicked();

protected:
    void mousePressEvent(QMouseEvent *event) override {
        QLabel::mousePressEvent(event);
        emit clicked();
    }
};

#endif // CLICKABLELABEL_HPP
```

```
#ifndef DESCRIPTORDETAILS_HPP
#define DESCRIPTORDETAILS_HPP

#include <QDialog>
#include <QString>
#include <QJsonArray>
#include <QLabel>

#include "descriptor.hpp"

namespace Ui {
class DescriptorDetails;
}

class DescriptorDetails : public QDialog
{
    Q_OBJECT

public:
    explicit DescriptorDetails(QWidget *parent = nullptr , bool access = false,QString Libr
aryPath = "");
    ~DescriptorDetails();
    void setDescriptor(Descriptor* descriptor);
    void setLibraryPath(QString libraryPath);
    QString getLibraryPath();
    bool access;

private slots:
    void on_filtreButton_clicked();
    void onFilterSelectionChanged(int index);
    void on_SaveChanges_clicked();
    void onLabelClicked(QLabel *clickedLabel);

private:
    Ui::DescriptorDetails *ui;
    Descriptor* currentDescriptor;
    QString LibraryPath;

};

#endif // DESCRIPTORDETAILS_HPP
```

```
#ifndef DESCRIPTOR_HPP
#define DESCRIPTOR_HPP
#include "image.hpp"
#include <QJsonObject>

class Descriptor {

private:
    unsigned int idDes;
    Descriptor* nextDescriptor;
    double cost;
    QString title;
    QString source;
    char access;
    Image image;

public:
    Descriptor(const Image& img);
    Descriptor(int idDesc, const Image& img);
    Descriptor(int idDesc, double costValue, const Image& img);
    Descriptor(int idDesc, double costValue, const QString& descTitle, const Image& img);
    Descriptor(int idDesc, double costValue, const QString& descTitle, const QString& descS
ource, const Image& img);
    Descriptor(int idDesc, double costValue, const QString& descTitle, const QString& descS
ource, const char descAccess, const Image& img);

    unsigned int getIdDescriptor() const;
    double getCost() const;
    QString getTitle() const;
    QString getSource() const;
    char getAccess() const;
    Image getImage() const;
    Descriptor* getNextDescriptor() const;

    void setIdDescriptor(int newIdDes);
    void setCost(double newCost);
    void setTitle(const QString& descTitle);
    void setSource(const QString& descSource);
    void setAccess(const char& descAccess);
    void setImage(const Image& img);
    void setNextDescriptor(Descriptor* nextDesc);

    void display() const;
    QPixmap cvMatToQPixmap(const cv::Mat& mat) const;
    cv::Mat QPixmapToCvMat(const QPixmap& pixmap) const;
    QJsonObject toJson() const;
};

#endif
```

```
#ifndef IMAGE_HPP
#define IMAGE_HPP

#include <opencv2/opencv.hpp>
#include <QString>
#include <QPixmap>
class Image {
public:
    Image(const QString& imgPath);

    void loadImage(const QString& imgPath);
    double calculateCompressionRatio(const QString& imgPath) const;
    void showImage(const QString& imgPath) const;

    QString getFormat() const;
    QString getPath() const;
    double getCompressionRatio() const;
    int getId() const;

    void setPath(const QString& newPath);
    void setId(const int newID);
    cv::Mat getContent() const;
    QPixmap getPixmap() const;



private:
    QString path;
    QString format;
    double compressionRatio;
    int idImage;
    cv::Mat content;
};

#endif // IMAGE_HPP
```

```
#ifndef IMAGEPROCCESSING_HPP
#define IMAGEPROCCESSING_HPP
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

class ImageProccessing
{
public:
    ImageProccessing();

    Mat calculateHistogram(const Mat& inputImage);
    Mat applyGaussianFilter(const Mat& inputImage);
    Mat toGrayScale(const cv::Mat& inputImage) ;
    Mat applyCustomMedianFilter(const cv::Mat& inputImage, int kernelSize);
    Mat applyEdgeDetection(const Mat& inputImage);
    Mat applyThreshold(const Mat& inputImage, int thresholdValue);
    Mat rotateImage(const Mat& inputImage, int angle);
    Mat applySIFT(const Mat& inputImage);
    Mat applyErosion(const Mat& inputImage, int kernelSize) ;
};

#endif // IMAGEPROCCESSING_HPP
```

```
#ifndef KERNELS_HPP
#define KERNELS_HPP

#include <vector>
using namespace std;

// Noyau Sobel pour le gradient X
const vector<vector<float>> SOBEL_X = {
    {-1, 0, 1},
    {-2, 0, 2},
    {-1, 0, 1}
};

// Noyau Sobel pour le gradient Y
const vector<vector<float>> SOBEL_Y = {
    {-1, -2, -1},
    { 0,  0,  0},
    { 1,  2,  1}
};

#endif // KERNELS_HPP
```

```
#ifndef LIBRARYMANAGEMENT_HPP
#define LIBRARYMANAGEMENT_HPP

#include <QString>
#include <math.h>
#include "descriptor.hpp"

using namespace std;


class ManageLibrary {

private:
    int acces;
    Descriptor* head ;
    QString libraryPath;


public:
    ManageLibrary(int acces, Descriptor* head,QString libraryPath);

    Descriptor* getDescriptor(unsigned int idDesc) const ;
    int getAcces() const ;

    void addDescriptor()     const;
    void deleteDescriptor()  const;
    Descriptor* searchDescriptor(unsigned int id)  const;
    void sortDescriptors()   const;
    int totalDescriptors()   const;
    void dipslayAllimages()  const;
    void display()           const;
    double displayCost(unsigned int id)     const;
    void createCostSubList() const;
    Descriptor* getHead() const;
    void setHead(Descriptor* head);
    void deleteDescriptor(Descriptor* descriptorToDelete);
    double getMaxDescriptorCost();
    double getMinDescriptorCost();
    ManageLibrary orderDescriptorsByCostDescending();
    ManageLibrary orderDescriptorsByCostAscending();
    void insertDescriptorInOrder(ManageLibrary& library, Descriptor* descriptor);
    void insertDescriptorInOrderAscending(ManageLibrary& library, Descriptor* descriptor);

    Descriptor* getDescriptorsByMaxCost(double maxCost);
    QString getLibraryPath() const;
    void setLibraryPath(QString path);

    void deletDescriptorFromMemory(Descriptor* descriptorToDelete);
    void saveLibraryToJson(QString libraryName);

    Descriptor* getDescriptorsBetweenMaxMinCost(double maxCost, double minCost);



};

#endif
```

```
#ifndef LOGINWINDOW_HPP
#define LOGINWINDOW_HPP

#include <QDialog>
#include "user.hpp"

namespace Ui {
class LoginWindow;
}

class LoginWindow : public QDialog
{
    Q_OBJECT

public:
    explicit LoginWindow(User &user, QWidget *parent = nullptr);
    ~LoginWindow();

private slots:
    void on_LoginButton_clicked();
private:
    Ui::LoginWindow *ui;
    User &user; // Reference to the User object


};

#endif // LOGINWINDOW_HPP
```

```
#ifndef USER_HPP
#define USER_HPP
#include <QString>
#include "librarymanagement.hpp"
#include <QJsonArray>

class User
{
private:


public:
    bool access;
    QString libraryPath;
    User() : access("") {};
    User(bool access);
    ManageLibrary loadLibrary(const QString& path) const  ;
    QJsonArray loadLibraries(const QString& librariesFilePath);
    // create a new library method
    void createLibrary(QString libraryName);
    void deleteLibrary(QString libraryToDelete);
};

#endif // USER_HPP
```