ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ НЕФТИ И ГАЗА
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)
ИМЕНИ И.М. ГУБКИНА»

ФАКУЛЬТЕТ КОМПЛЕКСНОЙ БЕЗОПАСНОСТИ ТЭК
КАФЕДРА БЕЗОПАСНОСТИ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

**Лабораторная работа №6**

по дисциплине «Специализированные языки и технологии программирования»

на тему «Разработка приложения на QML»

Выполнил студент:
группы КА-22-06
Воронин Алексей Дмитриевич

Преподаватель:

Греков Владимир Сергеевич

Москва, 2025

Оглавление

**Цель работы**

Освоить основы разработки приложений с использованием QML и Qt Quick. Создать мобильное или десктопное приложение с графическим пользовательским интерфейсом, реализованным на QML.

**Задание**

Разработка приложения "Погодный информатор".

**ЧАСТЬ 1 - Детальные инструкции к выполнению**

Создайте новый проект в Qt Creator, выбрав тип проекта "QML-приложение (Qt Quick)".

С использованием QML создайте интерфейс приложения, который должен включать:

- Элемент для отображения текущего города и погоды (температуры, состояния погоды, влажности, ветра).
- Поле для ввода названия города.
- Кнопку для обновления информации о погоде в выбранном городе.

Добавьте в Main.qml следующее:

```
import QtQuick 2.15
import QtQuick.Controls 2.15
import QtQuick.Layouts 1.15
import QtCore
import "WeatherFetcher.js" as WeatherFetcher

ApplicationWindow {
    visible: true
    width: 400
    height: 600
    title: qsTr("Погодный информатор")

    Settings {
        id: appSettings
        property string lastCity: ""
    }

    Rectangle {
        anchors.fill: parent
        color: "#f0f0f0"

        ColumnLayout {
            anchors.centerIn: parent
            spacing: 15
```

```
TextField {
    id: cityInput
    placeholderText: qsTr("Введите город")
    Layout.preferredWidth: 300
    font.pointSize: 16
    background: Rectangle {
        color: "#ffffff"
        radius: 5
        border.color: "#cccccc"
    }
}

Button {
    text: qsTr("Обновить погоду")
    font.pointSize: 16
    Layout.preferredWidth: 300
    background: Rectangle {
        color: "#4caf50"
        radius: 5
    }
    onClicked: {
        WeatherFetcher.fetchWeather(cityInput.text)
        appSettings.lastCity = cityInput.text
    }
}

ColumnLayout {
    id: weatherBlock
    spacing: 10
    opacity: 0.0

    Behavior on opacity {
        NumberAnimation { duration: 500 }
    }

    Text {
        id: cityName
        text: qsTr("Город: -")
        font.pointSize: 18
        font.bold: true
        color: "#333"
```

```
            }

            Text {
                id: temperature
                text: qsTr("Температура: -")
                font.pointSize: 18
                color: "#ff5722"
                font.bold: true
                Behavior on text {
                    NumberAnimation { duration: 300; easing.type: Easing.InOutQuad }
                }
            }

            Text {
                id: description
                text: qsTr("Описание: -")
                font.pointSize: 14
                color: "#757575"
            }

            Text {
                id: humidity
                text: qsTr("Влажность: -")
                font.pointSize: 14
                color: "#757575"
            }

            Text {
                id: wind
                text: qsTr("Ветер: -")
                font.pointSize: 14
                color: "#757575"
            }
        }

        Text {
            id: errorMessage
            color: "red"
            font.pointSize: 12
            wrapMode: Text.Wrap
```

```
            visible: false
            width: parent.width * 0.8
            horizontalAlignment: Text.AlignHCenter
        }
    }

    Component.onCompleted: {
        if (appSettings.lastCity !== "") {
            cityInput.text = appSettings.lastCity
            WeatherFetcher.fetchWeather(appSettings.lastCity)
        }
    }
  }
}
```

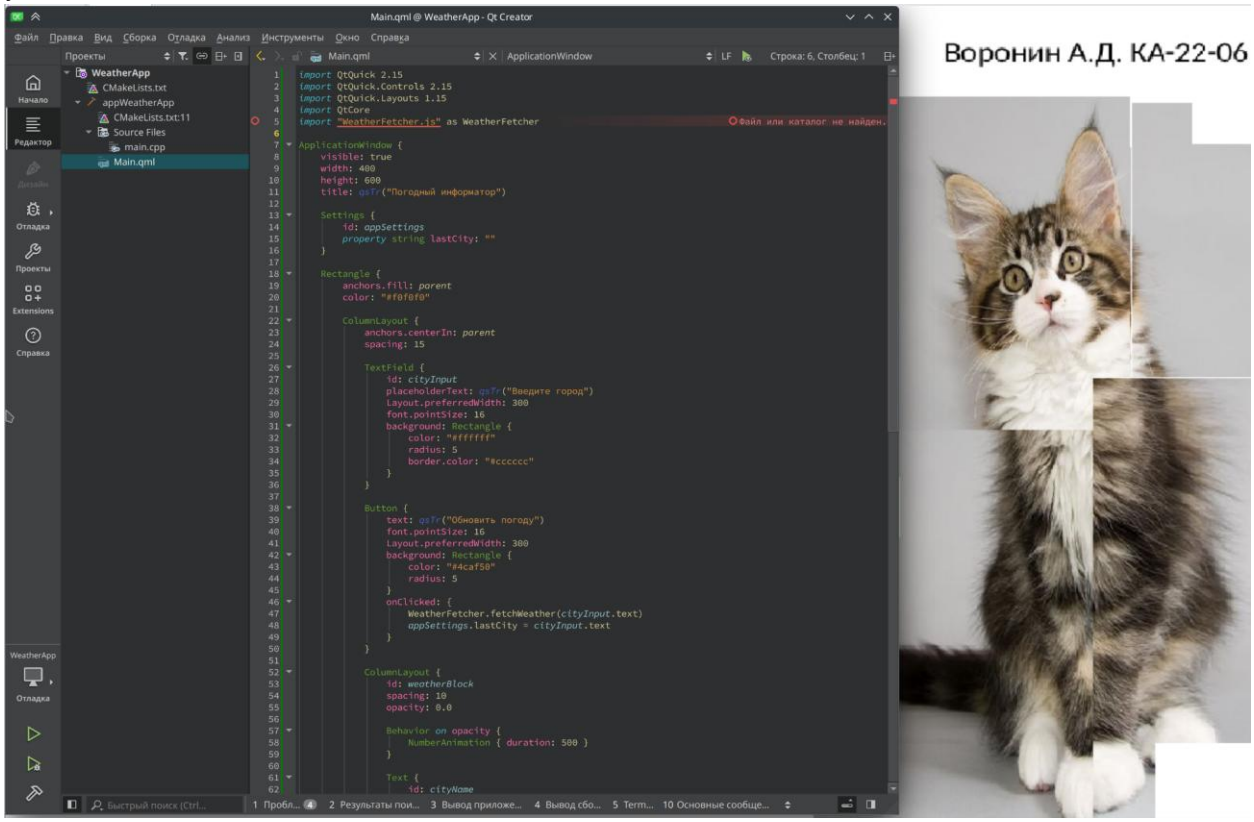


Рисунок 1 – редактирование Main.qml

Используйте JavaScript для обработки событий интерфейса (например, нажатие на кнопку обновления погоды).

Реализуйте запрос к погодному API (например, OpenWeatherMap) для получения актуальных данных о погоде. Для запросов используйте XmlHttpRequest или другие доступные средства QML для работы с сетью.

Создайте файл JS.

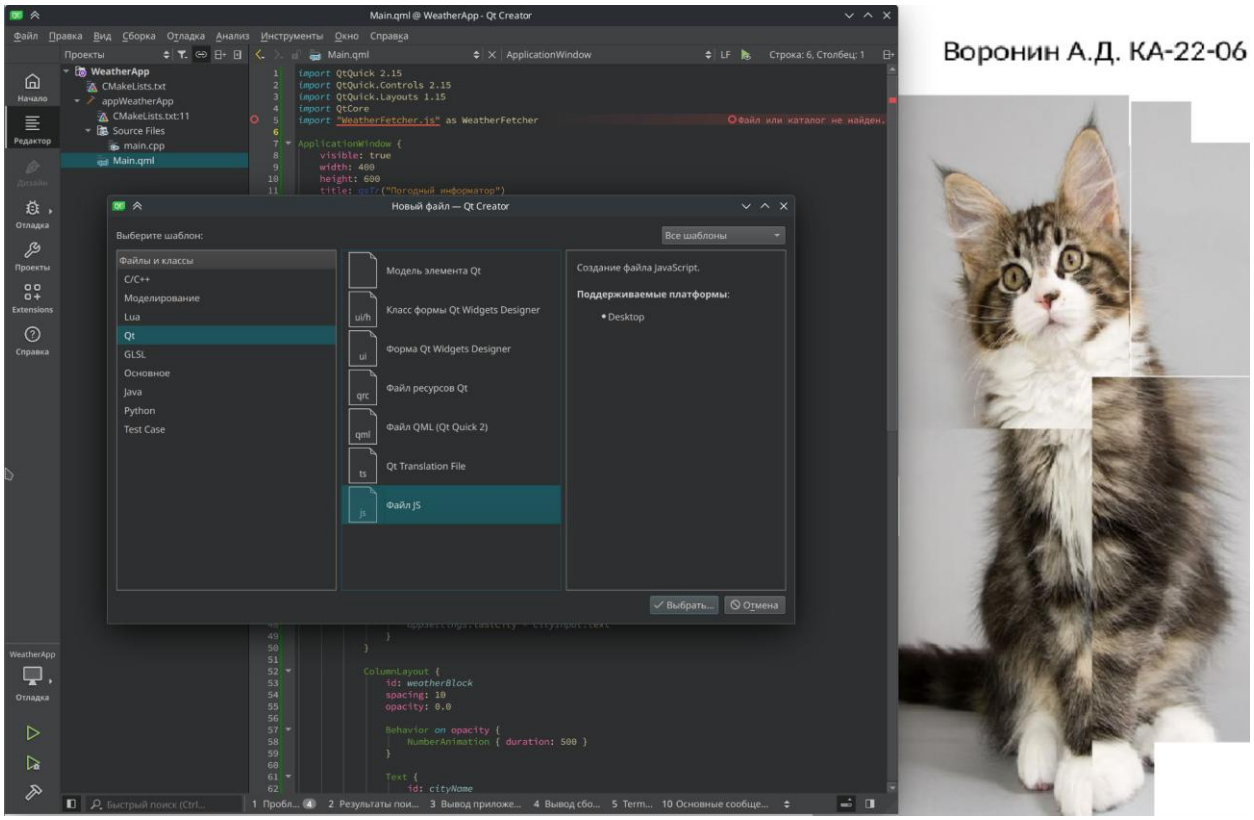


Рисунок 2 – Создание файла js

Назовите файл WeatherFetcher.js

Добавьте в WeatherFetcher.js следующее:

```
var apiKey = "53f2e1b43a53b9ec86018cdf1e816cfa"
var baseUrl = "https://api.openweathermap.org/data/2.5/weather"

function fetchWeather(city) {
    if (!city) {
        errorMessage.text = qsTr("Введите город")
        errorMessage.visible = true
        return
    }

    var xhr = new XMLHttpRequest()
    var url = baseUrl + "?q=" + encodeURIComponent(city) +
            "&appid=" + apiKey + "&units=metric&lang=ru"

    xhr.onreadystatechange = function () {
        if (xhr.readyState === XMLHttpRequest.DONE) {
            if (xhr.status === 200) {
```

```
                var response = JSON.parse(xhr.responseText)
                cityName.text = qsTr("Город: ") + response.name
                temperature.text = qsTr("Температура: ") + response.main.temp +
"°C"
                description.text = qsTr("Описание: ") +
response.weather[0].description
                humidity.text = qsTr("Влажность: ") + response.main.humidity + "%"
                wind.text = qsTr("Ветер: ") + response.wind.speed + " м/с"
                errorMessage.visible = false
                weatherBlock.opacity = 1.0
            } else {
                errorMessage.text = qsTr("Ошибка: проверьте название города")
                errorMessage.visible = true
                weatherBlock.opacity = 0.0
            }
        }
    }

    xhr.open("GET", url)
    xhr.send()
}
```

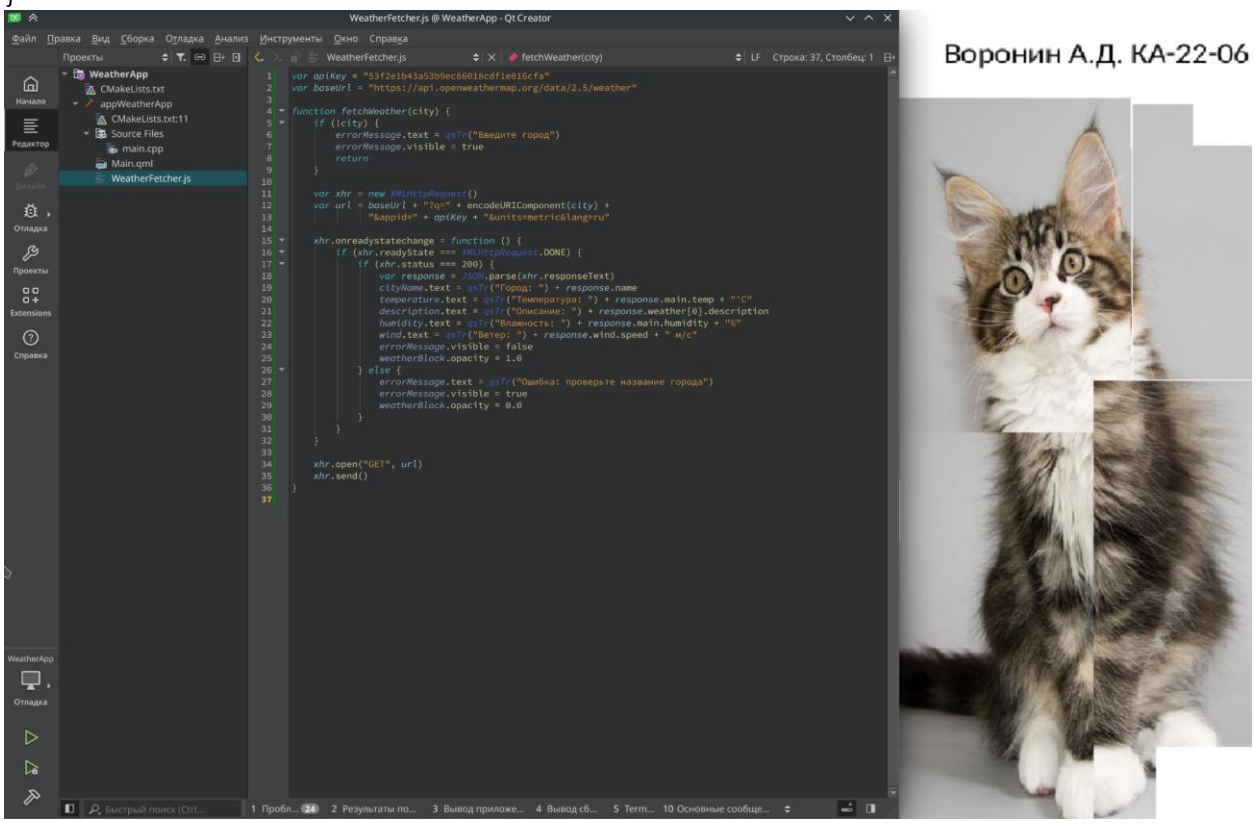


Рисунок 3 – Редактирование файла WeatherFtcher.js

После получения данных от API отобразите информацию о погоде в соответствующих элементах интерфейса.

Обработайте возможные ошибки запроса (например, неверное название города или проблемы с сетью) и информируйте пользователя об этих ошибках.

Добавьте в `main.cpp` следующее:

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QTranslator>
#include <QLocale>

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QCoreApplication::setOrganizationName("MyCompany");
    QCoreApplication::setApplicationName("WeatherApp");

    QTranslator translator;
    if (translator.load(QLocale(), "weatherapp", "_", ":/i18n")) {
        app.installTranslator(&translator);
    }

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("WeatherApp/Main.qml")));
    if (engine.rootObjects().isEmpty())
        return -1;

    return app.exec();
}
```

**ЧАСТЬ 2 - Дополнительные возможности**

**Кастомизация интерфейса**: Используйте возможности QML для создания анимаций, переходов и других эффектов, чтобы сделать интерфейс пользователя более динамичным и приятным в использовании.

**Локализация**: Добавьте поддержку нескольких языков в ваше приложение, позволяя пользователю выбирать предпочитаемый язык интерфейса.

**Сохранение настроек**: Реализуйте возможность сохранения последнего запрошенного города в локальном хранилище, чтобы при следующем запуске приложения информация автоматически обновлялась для этого города.

Все это проделано выше.

**ЧАСТЬ 3 - Тестирование приложения**

Соберем и запустим приложение.

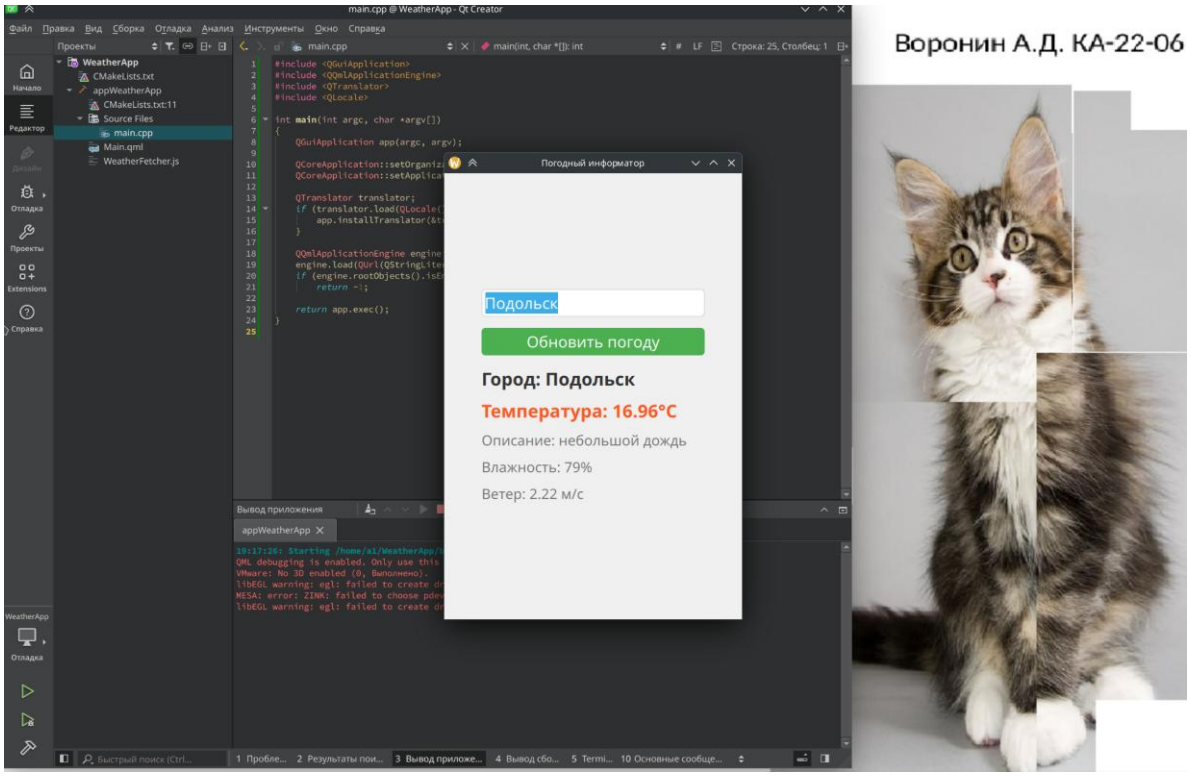


Рисунок 4 – Тестирование

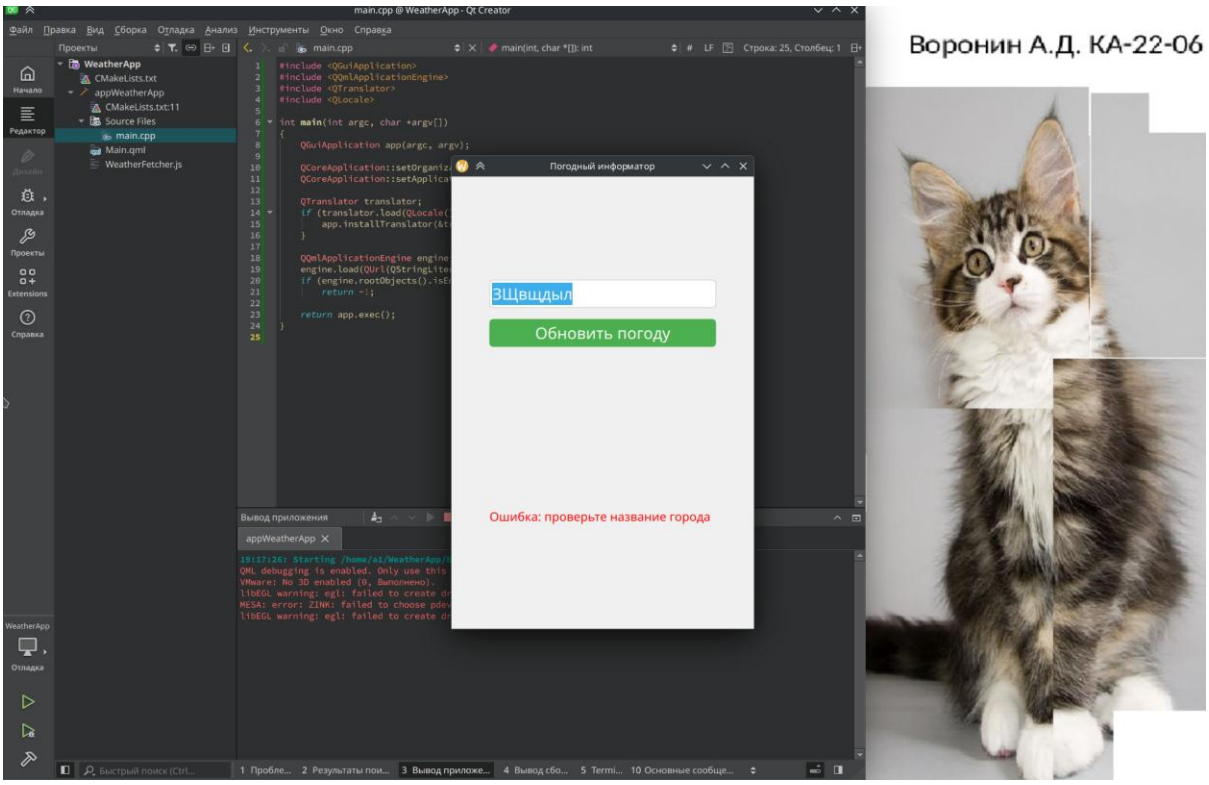


Рисунок 5 – Тестирование

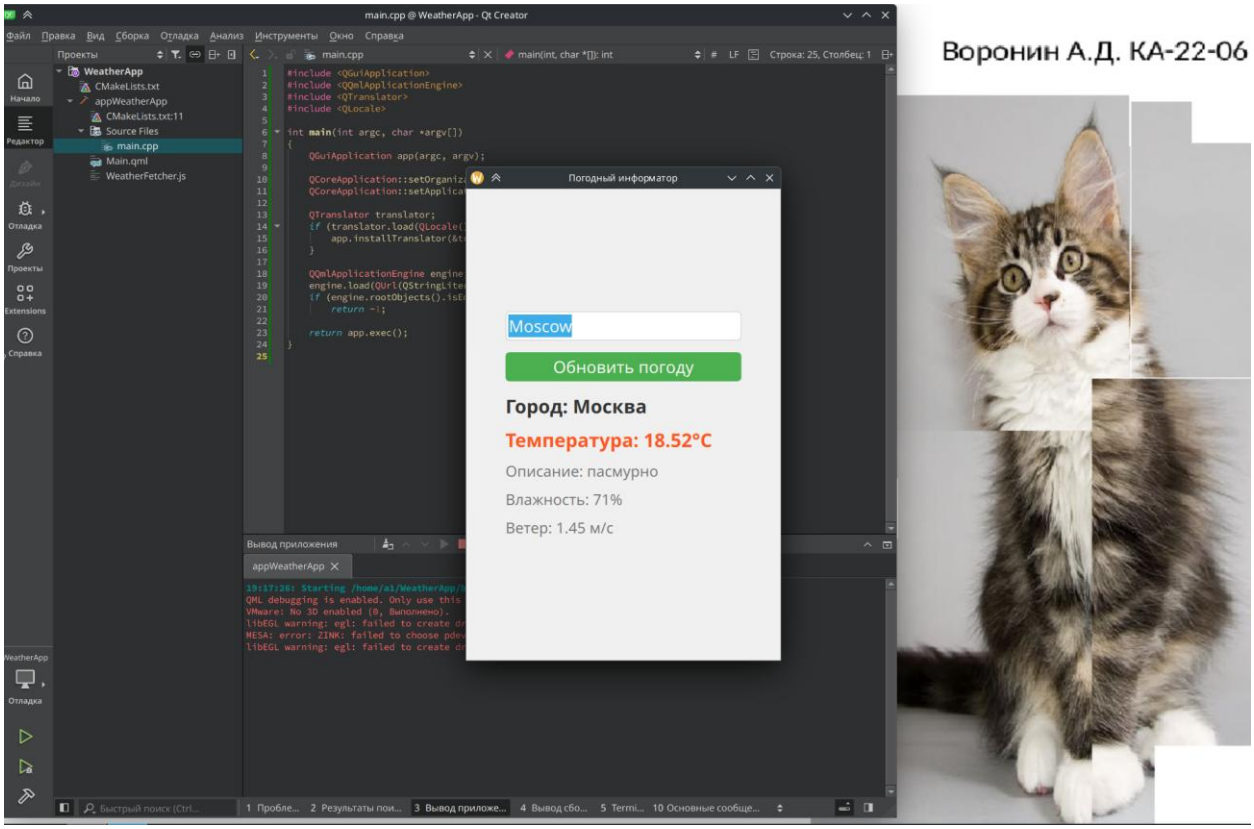


Рисунок 6 – Тестирование

**Самостоятельная работа**

Задание для самостоятельной работы

- Модифицируйте приложение так, чтобы оно не только показывало текущую погоду, но и предоставляло краткий прогноз на несколько дней вперёд.
- Реализуйте возможность выбора единиц измерения температуры (Цельсий, Фаренгейт).
- Создайте график изменения температуры и других погодных параметров (например, влажности и ветра) на протяжении дня.

Результат выполнения самостоятельной работы представлен на рисунке ниже.

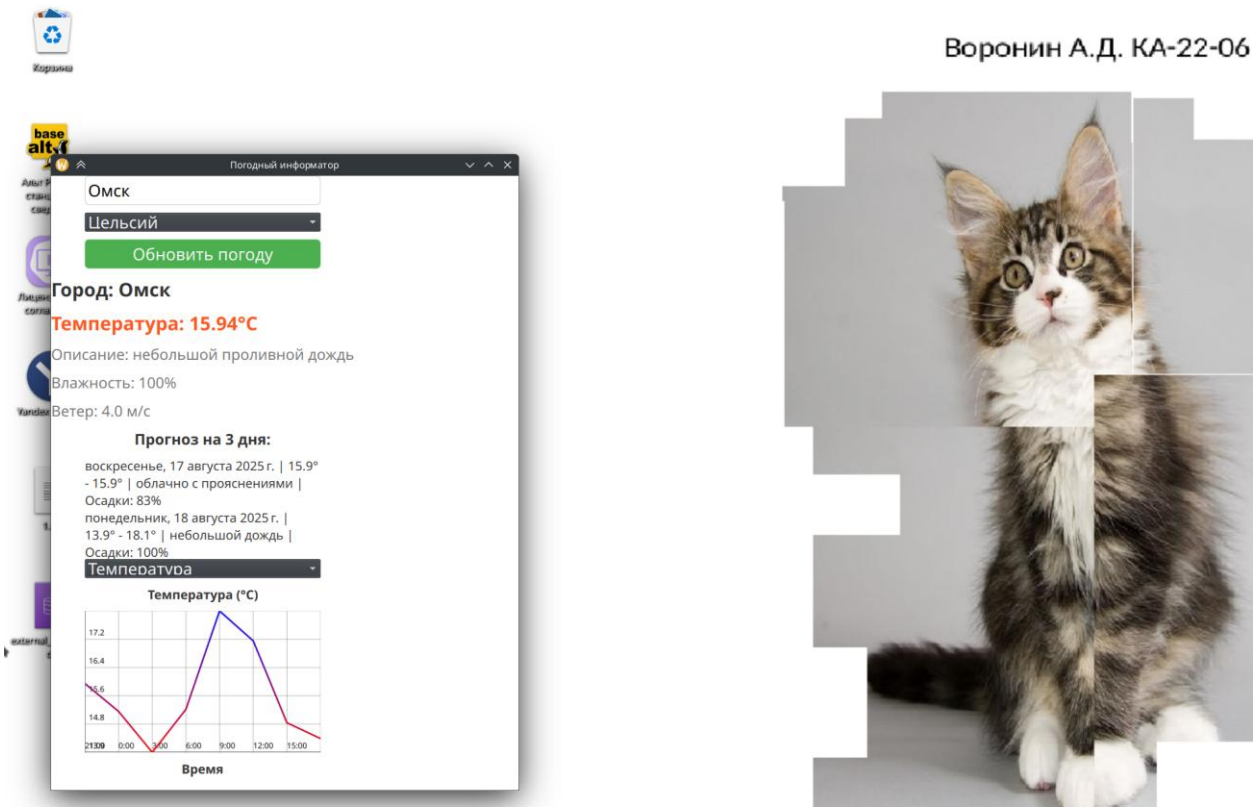


Рисунок 7 – Результат выполнения самостоятельной работы

## ЗАКЛЮЧЕНИЕ

В заключение, подводя итог проделанной работе, все поставленные в начале цели и задания данной лабораторной работы были достигнуты в полном объеме.

**Контрольные вопросы**

**1. Что такое QML и как оно связано с Qt Quick?**

QML (Qt Modeling Language) — это декларативный язык программирования, предназначенный для разработки пользовательских интерфейсов.

Qt Quick — это технология, которая обеспечивает реализацию QML приложений.

**2. Как осуществляется взаимодействие между QML и C++ в приложении?**

Взаимодействие между QML и C++ в приложении осуществляется с помощью нескольких встроенных механизмов. Чаще всего C++ используется для реализации бизнес-логики и работы с данными, а QML — для построения интерфейса. Чтобы связать их, классы C++ наследуются от QObject, описывают свойства с помощью Q_PROPERTY и методы с макросом Q_INVOKABLE или как публичные слоты. Эти классы можно зарегистрировать в QML через qmlRegisterType, либо передать их экземпляры напрямую в контекст с помощью setContextProperty. В QML после этого можно обращаться к их свойствам, вызывать методы и подписываться на сигналы.

**3. Как в QML использовать локализацию (поддержку нескольких языков)?**

1) В проекте создаются .ts файлы для каждого языка (например, app_ru.ts, app_en.ts). Эти файлы генерируются с помощью утилиты lupdate, которая автоматически извлекает все строки, помеченные для перевода.

2) В QML любая строка, которую нужно перевести, оборачивается в функцию qsTr():

3) После создания .ts файлов их нужно перевести вручную (например, через Qt Linguist). Переводы компилируются в бинарные .qm файлы с помощью утилиты lrelease.

4) В C++ коде создаётся объект QTranslator, который загружает нужный .qm файл в зависимости от выбранного языка.

**4. Что такое QtCore?**

QtCore — это один из основных модулей библиотеки Qt, который предоставляет базовый функционал, необходимый для работы практически любого приложения на Qt.