**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра технологий программирования**

**АНАЛИЗ ОБЛАЧНЫХ СЕРВИСОВ  
 ДЛЯ ПРОЕКТИРОВАНИЯ IoT-ПРИЛОЖЕНИЯ**

Курсовой проект

Левоцкого Глеба Сергеевича

Студента 3 курса

Специальность   
«прикладная информатика»

Научный руководитель:

Старший преподаватель

Давидовская М.И

Минск 2020

**РЕФЕРАТ**

Курсовой проект, с., 12 рис., 1 таблица.

**Ключевые слова:** IOT, GOOGLE IOT CORE, AZURE IOT SUITE, АНАЛИЗ ОБЛАЧНЫХ СЕРВИСОВ, УСТРОЙСТВА.

**Объект исследования —** объектом исследования являются облачные сервисы Google IoT core и Azure IoT suite и использование их возможностей для создания IoT-приложений. Так же анализируется API, предоставляемое для разработки приложений под конкретную платформу от конкретного сервиса. В качестве предмета исследования выбираем разработку приложения по сбору информации о температурном состоянии и исследование возможностей изучаемых облачных сервисов.

**Цели работы —** рассмотреть методы создания виртуальных устройств для облачных сервисов, а также спроектировать приложения, симулирующие работу устройств. Продемонстрировать базовые навыки взаимодействия между устройствами, платформой и сервером. Разработать простейшие приложения, воспользовавшись полученными знаниями.

**Методы исследования —** а) теоретические: изучение статей, посвященных решению различных задач с использованием облачных сервисов, а так же интернет ресурсов и книг, посвященных разработке IoT приложений; б) практические: проектирование приложений, симулирующих виртуальные устройства путем обобщения опыта проработки примеров официальных поставщиков сервисов, а так же образовательных практических примеров из сети интернет.

**Результатами являются —** приложения, симулирующие виртуальные устройства на подобие настоящих элементов IoT, а таже результат работы этих приложений и пошаговое описание создания базовых модулей взаимодействия системы IoT.

**Область применения** — изучение и создание собственных приложений симуляции элементов IoT на основе полученных знаний с использованием базовых элементов, продемонстрированных в ходе работы.

**РЭФЕРАТ**

Курсавы праект, с., 12 мал., 1 табліца.

**Ключавыя словы:** IOT, GOOGLE IOT CORE, AZURE IOT SUITE, АНАЛІЗ ВОБЛАЧНЫХ СЭРВІСАЎ, ПРЫЛАДЫ.

**Аб’ект даследвання —** аб’ектам даследвання з’яўляюцца воблачныя сэрвісы Google IoT core і Azure IoT suite і выкарыстанне іх магчымасцяў для стварэння IoT-прылад. Таксама аналізуецца API, якое падаецца для распрацоўкі прыкладанняў пад пэўную платформу пэўнага сэрвіса. У якасці прадмета даследвання абрана распрацоўка прыкладання па збору інфармацыі аб тэмпературным становішчы і даследванне магчымасцяў вывучыемых воблачных сэрвісаў.

**Мэты працы —** разглядзець метады стварэння віртуальных прылад для воблачных сэрвісаў, а таксама спраектаваць прыкладанні, якія сімулююць працу прылад. Прадэманстраваць базавыя магчымасці ўзаемадзеяння паміж прыладамі, платформай і серверам. Распрацаваць найпростыя прыкладанні з дапамогай атрыманых тэарэтычных ведаў.

**Метады даследвання —** а) тэарэтычныя: вывучэнне артыкулаў, якія асвятляюць вырашэнне разнастайных задач з выкарыстаннем воблачных сэрвісаў, а таксама інтэрнэт-рэсурсаў ды кніг, якія прысведчаны распрацоўке IoT прыкладанняў; б) практычныя: праектаванне прыкладанняў, якія сімулююць віртуальныя прылады праз абагульненне досведу разгляду прыкладаў афіцыйных распрацоўнікаў сэрвісаў, а таксама адукацыйных практычных прыкладаў з сеткі Інтэрнэт.

**Вынікамі з’яўляюцца —** прыкладанні, сімулюючыя віртуальныя прылады накшталт сапраўдных элементаў IoT, а таксама вынік працы гэтых прыкладанняў і пакрокавае апісанне стварэння базавых модуляў узаемадзеяння сістэмы IoT.

**Вобласць ужывання** — вывучэнне і стварэнне ўластных прыкладанняў сімуляцыі элементаў IoT на выснове атрыманых ведаў з выкарыстаннем базавых элементаў, прадэманстраваных цягам працы.

**ESSAY**

COURSE project, s., 12 pic., 1 table.

**Keywords:** IOT, GOOGLE IOT CORE, AZURE IOT SUITE, ANALYSIS CLOUD SERVICES, DEVICE.

**The object of research** is the cloud services Google IoT core and Azure IoT suite and the usage of their capabilities for IoT applications creation. We also analyze the API provided for developing applications for a specific platform from a specific service. As a subject of research, we choose the development of an application for collecting information about the temperature state and researching the capabilities of the studied cloud services.

**The objects of the work** are to consider methods of creating virtual devices for cloud services, as well as to design applications that simulate the operation of devices. Demonstrate basic communication abilities between devices, platform and server. Develop simple applications using the gained knowledge.

**Research methods** — a) theoretical: study of articles devoted to solving various problems using cloud services, as well as Internet resources and books on the development of IoT applications; b) practical: designing applications that simulate virtual devices by summarizing the experience of working out examples of official service providers, as well as educational practical examples from the Internet.

**The results are** applications that simulate virtual devices similar to real IoT elements, as well as the result of the operation of these applications and a step-by-step description of the creation of basic interaction modules of the IoT system.

**Scope** — explore and build your own IoT element simulation applications based on the gained knowledge using the basic elements demonstrated during the work.

Оглавление

[ВВЕДЕНИЕ 6](#_Toc58502318)

[ГЛАВА 1 ОБЛАЧНЫЕ СЕРВИСЫ ДЛЯ ИНТЕРНЕТА ВЕЩЕЙ 8](#_Toc58502319)

[1.1 IoT-решения Microsoft на примере Azure IoT Suite 8](#_Toc58502320)

[1.2 IoT-решение Google на примере Google IoT Core 10](#_Toc58502321)

[ГЛАВА 2 ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ ИНТЕРНЕТА ВЕЩЕЙ 13](#_Toc58502322)

[2.1 Проектирование приложения для облачного сервиса Microsoft Azure IoT 14](#_Toc58502323)

[2.2 Проектирование приложения для облачного сервиса Google Cloud Core 17](#_Toc58502324)

[ГЛАВА 3 РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЙ ИНТЕРНЕТА ВЕЩЕЙ 21](#_Toc58502325)

[3.1 Разработка приложения для облачного сервиса Microsoft Azure IoT. 21](#_Toc58502326)

[3.2 Разработка приложения для облачного сервиса Google IoT Core 35](#_Toc58502327)

[Заключение 51](#_Toc58502328)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 53](#_Toc58502329)

[Приложения 54](#_Toc58502330)

[Приложение А 54](#_Toc58502331)

[Приложение Б 55](#_Toc58502332)

[Приложение В 56](#_Toc58502333)

ВВЕДЕНИЕ

Каждый из нас хоть однажды видел, слышал или думал об умном доме, где каждый предмет подключен к общей сети и вместе они облегчают повседневную жизнь. Включить или выключить чайник не вставая с дивана сделать кофе, поставить стираться загруженное белье, выключить свет в соседней комнате, покормить кота и многое-многое другое. Все это позволяет сделать IoT. Достаточно лишь иметь доступ в интернет, и сможем управлять своим домом из любой точки земного шара. Получить извещение пожарной тревоги, проверить температуру холодильника — пользователь может получать всю информацию о состоянии устройств в любой момент времени.

В современном мире прогресс не стоит на месте. Вокруг нас постоянно появляются новые устройства, которые еще пять лет назад казались нам чем-то недоступным. Появляются устройства, которые помогают нам в повседневной жизни. У многих дома есть автоматические устройства, которые требуют единожды настроить их, а далее они будут сами выполнять нужную работу. Современные компании предоставляют еще и клиентские приложения, благодаря которым появляется возможность из любого места контролировать состояние устройства и спланировать своевременное ТО.

Одним домом не стоит ограничиваться. К сферам, где могут применяться устройства «интернета вещей», относятся парковки, система гостиничных номеров, фермерские системы контроля и многое-многое другое. В эпоху глобального покрытия поверхности планеты интернетом, это направление начинает завоевывать все больше и больше рынка, предлагая собственные оригинальные решения различных проблем контроля и управления системами.

Разрабатываемое приложение на своем примере продемонстрирует работу одного из самых популярных элементов умного дома — термометр. Этот элемент является одним из важнейших в каждом доме. Возвращаясь домой с учебы/работы или других мероприятий, каждому хочется чувствовать себя дома комфортно, а одним из базовых элементов является соответствующая температура. Получив информацию от термометра, пользователь может включить отопление либо же кондиционер, если температура нас не устраивает.

Приложения разрабатываются для удовлетворения потребностей пользователя, удобства использования и сокращения временных трат обладателя. Качественное приложение, нацеленное на дружеское взаимодействие с пользователем является немалой частью разработки проекта, ведь удобство использования не только сокращает временные затраты пользователя, но и вселяет желание пользоваться им снова и снова. Одними из самых крупных компаний, предоставляющих удобную, а самое главное, качественную платформу по разработке современной IoT системы являются корпорации Google и Microsoft. Каждая из них предоставляет конкурентно сильный продукт, а именно Google Cloud IoT и Azure IoT Suite.

Объектом исследования являются сервисы, предлагающие систему регистрации объектов конкретной IoT—системы, а также системы анализа данных, получаемых от устройств. Каждая из платформ имеет собственный подход, являющийся схожим с остальными, но имеющий, как и любая другая платформа, отличительные черты и тонкости использования при разработке.

Теоретическим методом исследования является изучения статей и материалов, посвященных проблемам, а также проработке базовых элементов для создания собственных IoT-приложений и систем. Практическим методом исследования является написание собственных устройств, подключений, а также обработки уведомлений от каждого из созданных устройств.

В первой главе рассматриваются сервисы для устройств «интернета вещей», предоставляемые одними из самых известных компаний. Таковыми являются Google и Microsoft. Данные компании предоставляют платформы для разработки IoT систем, богатые дополнительными сервисами, ускоряющими и облегчающими разработку проектов. Корпорация Microsoft предлагает решение на основе Azure IoT Suite. Google же, в свою очередь, предлагает решение на основе платформы Google IoT Core. Обе системы обладают широкими возможностями, а также содержат множество примеров и хорошую документацию, что является большим плюсом для начинающих разработчиков.

Вторая глава посвящена проектированию приложений на основе описанных в первой главе платформ. Во второй главе приведены подробные описания планируемых систем, а также отражены все особенности создания и сопровождения системы. На платформе компании Microsoft спроектирована базовая система отправки и получения сообщений, а вот на платформе Google IoT Core задача усложнена путем добавления изменения настроек конфигурации устройств.

Третья глава полностью посвящена разработке обоих приложений. В мельчайших подробностях описан каждый шаг разработки, от создания приложения и подключения нужных библиотек, запуска приложений. Описаны используемые все API и библиотеки. Приложение для Azure IoT Suite реализовано на родном языке компании Microsoft — C#, а вот написание приложения для платформы Google IoT Core реализовано на языкe Python. В конце главы приведены результаты работы обоих приложений.

ГЛАВА 1  
ОБЛАЧНЫЕ СЕРВИСЫ ДЛЯ ИНТЕРНЕТА ВЕЩЕЙ

1.1 IoT-решения Microsoft на примере Azure IoT Suite

Microsoft предлагает целую экосистему по созданию облачных сервисов и приложений. Microsoft Azure IoT Suite — это комплексное решение со встроенными инструментами безопасности, которые обеспечивают защиту приложений, от базовых облачных серверов баз данных, до виртуальных машин и интернета вещей, на всех уровнях. Одним из таких решений является служба «Центр IoT», предоставляющая такие возможности, как создание и регистрация устройств, анализ обращений к сервису, который визуализируется путем построения графиков активности, и многое другое.

Любой центр IoT имеет подписку, характеризующую возможности данного центра. Возможности варьируются от числа подключенных устройств, до максимального числа получаемых и отправляемых сообщений центром. Подписка имеет собственный идентификатор, характеризующий уникальность оплаченной подписки. Кроме подписки, центр IoT имеет Имя узла (Hostname), состояние (Active / Disabled), текущее расположение сервера, а также теги.

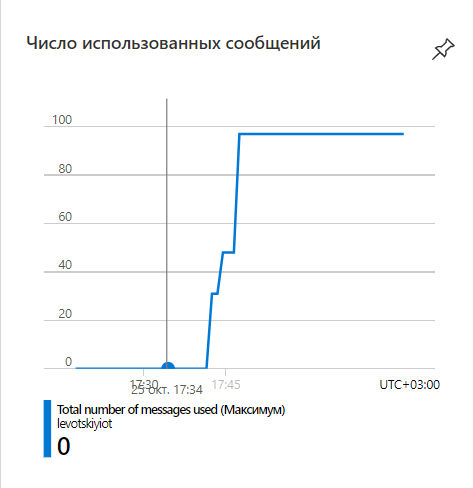
 Для анализа действий, проведенных с сервисом, Microsoft Azure IoT Center предлагает графики самых популярных запросов по получению информации о действиях, таких как число использованных сообщений сервиса, сообщений от устройств к облаку, число подключенных устройств, а также общее число устройств Интернета вещей. Пример одного из этих графиков можно увидеть на рисунке 1.1.

Рисунок 1.1 — График числа использованных сообщений.

Все данные, поступающие с устройств интернета вещей и отправляемые туда же, должны быть хорошо защищены, так как представляют из себя конфиденциальную информацию. Microsoft Azure IoT Center предлагает систему шифрования для защиты данных, устройств и подключений от злоумышленников. Система шифрования поддерживает несколько различных алгоритмов шифрования, таких, как AES и SHA-2.

IoT Center состоит из устройств и сервисов обработки информации, получаемых от этих устройств. Любое устройство имеет собственный идентификатор, представляющий из себя строковое имя, а также поля, отображающие устройство, такие как состояние устройства, время последнего обновления состояния, тип проверки подлинности и число сообщений, отправленных из облака на устройство.

Microsoft Azure IoT Center предоставляет графический интерфейс для системы управления устройствами Create Read Update Delete (далее CRUD), позволяющую в полной мере управлять устройствами, подключенными к данному IoT центру. Так же, помимо GUI, существует консольный вариант работы с Azure IoT Center. Данный вариант в полной мере удовлетворяет всем потребностям разработчика, а также не менее удобен.

Экосистема используется во многих средах жизни человека. Данная платформа нашла себя в отраслях здравоохранения и энергетики, Транспортировки и логистики, дискретного и непрерывного производства. Сама корпорация Microsoft предоставляет множество сервисов-посредников, внедрение который в проект на основе Azure IoT позволяет качественно улучшить разрабатываемый продукт. Так же, помимо сервисов, компания предоставляет пошаговое обучение разработке и предоставляет множество литературы, позволяющей не только понять, но и начать разрабатывать собственные системы.

В качестве API, для взаимодействия с системой компания Microsoft предоставляет целый набор пакетов под эгидой Microsoft.Azure для дочернего языка компании C# как для платформы .Net Framework, так и для .Net Core. Так же существуют пакеты для языков Java, JavaScript, Python. Все взаимодействие посредством API происходит по зашифрованным каналам, что обеспечивает достоверность получаемой информации и сохранность данных, отправленных системе.

1.2 IoT-решение Google на примере Google IoT Core

Корпорация Google предоставляет целую платформу по созданию, анализу и представлению IoT проектов. Данная платформа занимает одну из лидирующих позиций как по качеству предоставляемых сервисов и технологий, так и по удобству разработки новых систем. Для создания новой системы разработчику требуется создать новый проект и выдать ему имя. К выданному имени дописывается ключ. В совокупности, полученное имя является идентифицирующим для данного проекта. После создания нового проекта, разработчику доступно множество специализированных сервисов. Каждый из низ предоставляет решение какой-то определенной проблемы. Сервисы предоставляют не только обработку полученных данных, но также являются API всеми нами знакомых сервисов Google Maps, Gmail, Google Play и многих других. В библиотеке API с легкостью можно найти и подключить все необходимые сервисы и системы. Некоторые из них являются бесплатными, а некоторые предоставляются на платной основе. В данной библиотеке все сервисы подразделены на следующие категории:

* Карты
* Машинное обучение.
* G Suite. (Включает в себя работу с документами. Gmail, Drive и прочее.)
* YouTube.
* Социальные сети.
* Реклама.
* Мобильные решения.
* Google Cloud API.
* Другое.

Каждый проект, созданный на основе Google Cloud Platform обладает базовой панелью управления сервисами и API. На данной панели разработчик может наблюдать собранную статистику за последнее время. Выбор времени предоставляется в диапазоне от 1 часа, до 30 дней. В собранную статистику входят графики, отвечающие за визуализацию ошибок, медианной задержки, а также трафика, использованного за выбранный промежуток времени. Помимо графиков, разработчик может наблюдать таблицу с сервисами и статистикой, аналогичной с данными графиков, но только уже для выбранного сервиса. Платформа отличается большим удобством использования, что подтверждается возможностью скачать в CSV формате файл статистики как отдельно взятого графика, так и всю статистику в совокупности.

Для начинающих разработчиков платформа предоставляет множество руководств, нацеленных в кратчайшие сроки ввести читателя в курс дела и безболезненно освоить базовые навыки. Руководства начинаются от размещения простейшего приложения до подключения и использования ключевых элементов системы, таких как облачные хранилища, виртуальные вычислительные устройства и многое другое.

Для полноценного и качественного представления работы системы, а также сбора статистики и оптимизации работы проекта, Google Platform предоставляет полноценную систему мониторинга, позволяющую в полной мере оценить работу каждого элемента системы. Данная система не только предоставляет необходимые сведения, но и разделяет возможности оповещения, а также реализует добавление проверки работоспособности системы. Кроме всего вышеперечисленного, система мониторинга предоставляет конструктор по созданию запросов на вывод статистики. Данном конструктору передается тематика получаемых данных, а далее происходит уточнение. Так, на примере тематики «VM Instanse» можно получить данные по:

* CPU Usage — долю использованных возможностей виртуальной машины.
* CPU Utilization — частичное использование выделенного ЦП в данном проекте. Значения обычно представляют собой числа от 0,0 до 1,0 (но некоторые типы машин допускают разбиение выше 1,0). На диаграммах значения отображаются в процентах от 0% до 100% (или более). Эта метрика сообщается гипервизором виртуальной машины и может отличаться от значения, которое отображается внутри виртуальной машины.
* Disk read bytes — количество байт.
* Disk write bytes — количество байт записанных на диск.
* Received bytes — количество байт, полученных из сети.

Полученный запрос можно модифицировать, добавив фильтрацию по идентификатору проекта, идентификатору виртуальной машины, зоне расположения проекта и многому другому. Так же можно отсортировать полученные данные по тем же признакам, что описаны выше. Так же доступны агрегирующие функции (min, max, sum, count и многое другое). Помимо прочего, так же доступна регулировка временного периода, которая несомненно повышает удобство использования. Все вышеописанные возможности предоставляют мощнейший механизм выборки информации из имеющихся данных.

Платформа так же предоставляет систему обработки ошибок и создания отчетов о полученных ошибках. Изучить настройку данной систему можно в специальном руководстве, предоставляемом компании Google.

**ВЫВОДЫ**

Каждая из рассмотренных платформ обладает широким спектром возможностей и предоставляет удобное взаимодействие на протяжении всего этапа разработки. На начальных этапах системы очень схожи, но, чем сильнее разработчик углубляется в тонкости каждой из платформ, тем больше приходит понимание их различия.

ГЛАВА 2  
ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ ИНТЕРНЕТА ВЕЩЕЙ

Любое приложение интернета вещей базируется на множестве устройств, которые отслеживают окружающую среду и в определенный момент времени предоставляют информацию, получаемую из окружающей среды, и собственное состояние пользователю либо же серверу.

Выделяют несколько типов устройств:

1. Сенсоры (датчики)
2. Актуаторы
3. Шлюзы (гейты).

К сенсорам относятся обособленные устройства, имеющие от одного, но нескольких измерительных приборов, получающие информацию из окружающей среды и предоставляющие ее в отдельности, для каждого измерительного прибора, либо же в совокупности, как единое целое. Сенсор легко может иметь в своем устройстве промежуточные вычислительные, либо измерительные преобразователи. Примеры сенсоров: микрофон, камера, термометр, сенсорный экран, датчик движения и множество прочих, менее распространенных устройств.

К актуаторам относятся элементы автоматического управления, основной задачей которого воздействовать на элемент управления либо же окружающую среду. В большинстве своем, актуаторы состоят из 2 основных функциональных блоков. К первому блоку относят исполнительное устройство либо же механизм. Ко второму блоку относят регулирующую часть. Примерами могут выступать динамики, сервоприводы, механические устройства и прочее.

К шлюзам относят устройства, к которым подключены другие устройства интернета вещей и которые проводят поверхностный анализ информации, поступающей от подключенных к ним устройств. В роли шлюзов чаще всего используются микрокомпьютеры. Примерами являются Raspberry pi, Arduino и прочие.

Для того, чтобы построить систему мониторинга, достаточным условием является наличие сенсоров, удовлетворяющих условию поставленной задачи и сервера, который будет играть роль гейта, собирающего, анализирующего и отправляющего информацию на основной сервер для дальнейшего анализа собранных данных. Например, используя датчик движения и микрокомпьютер, например, Raspberry pi, можно организовать охранную систему, которая при срабатывании будет отправлять уведомление на сервер. Если к этой мониторинговой системе добавить актуатор в виде динамика, можно симулировать сигнал тревоги при каждом срабатывании датчика движения.

2.1 Проектирование приложения для облачного сервиса Microsoft Azure IoT

Первым шагом будет создание центра IoT в Microsoft Azure.

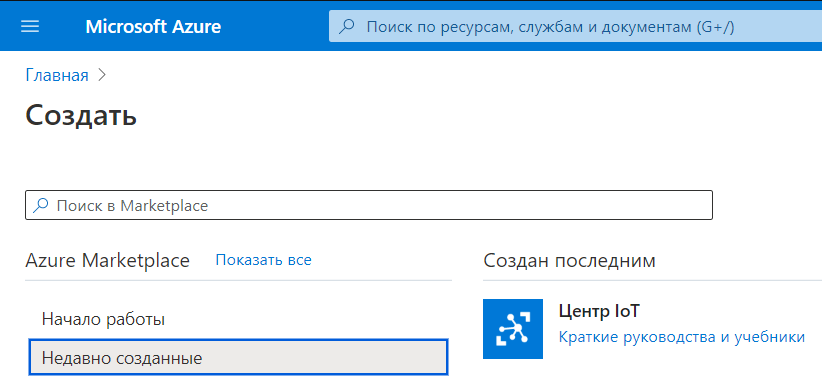


Рисунок 2.1.1 — создание центра IoT

Далее следует оформление подписки для данного центра и настройки его имени. Для демонстраций возможностей нам хватит бесплатной подписки, включающей себя возможность поддержки одного подключенного устройства и отправки не более 100 сообщений различного вида в сутки. Данная подписка создана для начинающих разработчиков, желающих познакомиться с системой и ее возможностями. Ресурсов, предоставляемых данной подпиской вполне должно хватить для начальных проектов и проверки взаимодействия разработанных компонентов.

Перед тем, как изменять центр IoT, следует продумать, как будет выглядеть конечный продукт, какие связи будут между модулями проекта и как они будут взаимодействовать. Для данной платформы будет разрабатываться система, по схеме producer/consumer. В роли producer будет выступать телеметрический датчик. Producer будет производить продукт в виде телеметрических данных. В роли Consumer будет выступать отслеживающее устройство. Consumer будет потреблять продукцию и выводить ее пользователю. Так как Microsoft предоставляет API для подключения и взаимодействия, то в данной ситуации вопросы синхронизации полностью ложатся/кладутся на их плечи. Producer будет выкладывать свою «продукцию» в топике, зарезервированном специально для устройства, которое он представляет, а consumer будет получать от туда же информацию, так как ему передается необходимый идентификационный ключ.

Представление разрабатываемой системы:

Рисунок 2.1.2 — схема приложения для платформы Azure IoT Suite

Первым шагом разработки будет создание макета устройства. Так как для данного сервиса проектируем приложение, симулирующее работу, то и устройство будет виртуальным. Это обозначает, что в роли устройства будет выступать процесс, отправляющий сигнал на сервер, а на перовых шагах проектирования у нас будет процесс, отлавливающий эти сигналы. Это может быть не обязательно процесс, находящийся на этом компьютере, это может быть процесс, выполняющийся в абсолютно другом месте, исполняемое устройство которого подключено к беспроводной сети Wi-Fi.

Устройство, которое планируется разработать, будет симулировать температурный датчик измерения температуры и влажности комнаты. Данное устройство будет через определенный промежуток времени, положим, 30 минут, отправлять на сервер свое текущее состояние, а именно температуру окружающей среды и влажность.

Следующим шагом будет регистрация устройства в центре IoT. Данный шаг будет так же выполнен программно, путем выдачи уникального имени устройству и подключения к центру IoT при помощи автогенерируемой, этим же центром, строки подключения для устройств. Эти возможности предоставляет нам Microsoft путем возможности специального API для подключения. Воспользовавшись им, можно с легкостью проделать описанные выше действия по подключению и соединению.

После успешной регистрации виртуального устройства под уникальным именем, следующим шагом станет разработка приложения, отвечающего за перехват сообщений, посылаемых центру IoT. Так как у нас будет всего одно устройство, то данное приложение не будет реализовывать распознавание устройства. Если это понадобится, то для этих целей существует система маршрутизации сообщений и для каждого устройства можно создать отдельную папку. В данном приложении будет продемонстрирована строка маршрутизации.

После разработки трех приложений:

1. Приложение для регистрации устройства и получения его уникального ключа.
2. Приложение для отправки данных на сервер (центр IoT).
3. Приложение для перехвата сообщений.

будут запущены в 2 процессах приложения по отправке данных и приложение по перехвату сообщений, и зафиксируем результат.

После того, как убедимся, что начальная система работает, а именно, приложение, перехватывающее сообщение, будет выводить те данные, которые телеметрический датчик передавал центру IoT, можно применять данные базовые операции в нужных нам приложениях. Продемонстрированная модель устройства является лишь симуляцией, но для демонстрации возможностей сервиса Azure IoT Suite достаточно. Для создания более сложных систем рекомендуется использовать исходный код данных систем регистрации, подключения, отправки и обработки сообщений.

2.2 Проектирование приложения для облачного сервиса Google Cloud Core

В данном параграфе рассмотрим процесс проектирования простой, но полноценной системы интернета вещей. В данной системе, в качестве устройств, будут выступать датчики температуры. Датчики будут посылать температурные сведения своим каналам телеметрии. В системе так же будет присутствовать сервер. Сервер будет управлять устройствами охлаждения, путем анализа телеметрических данных, полученных из облачного раздела Pub/Sub. По данным сведениям сервер будет решать, включать или выключать вентиляторы отдельных устройств с помощью обновления конфигурации Cloud IoT Core.

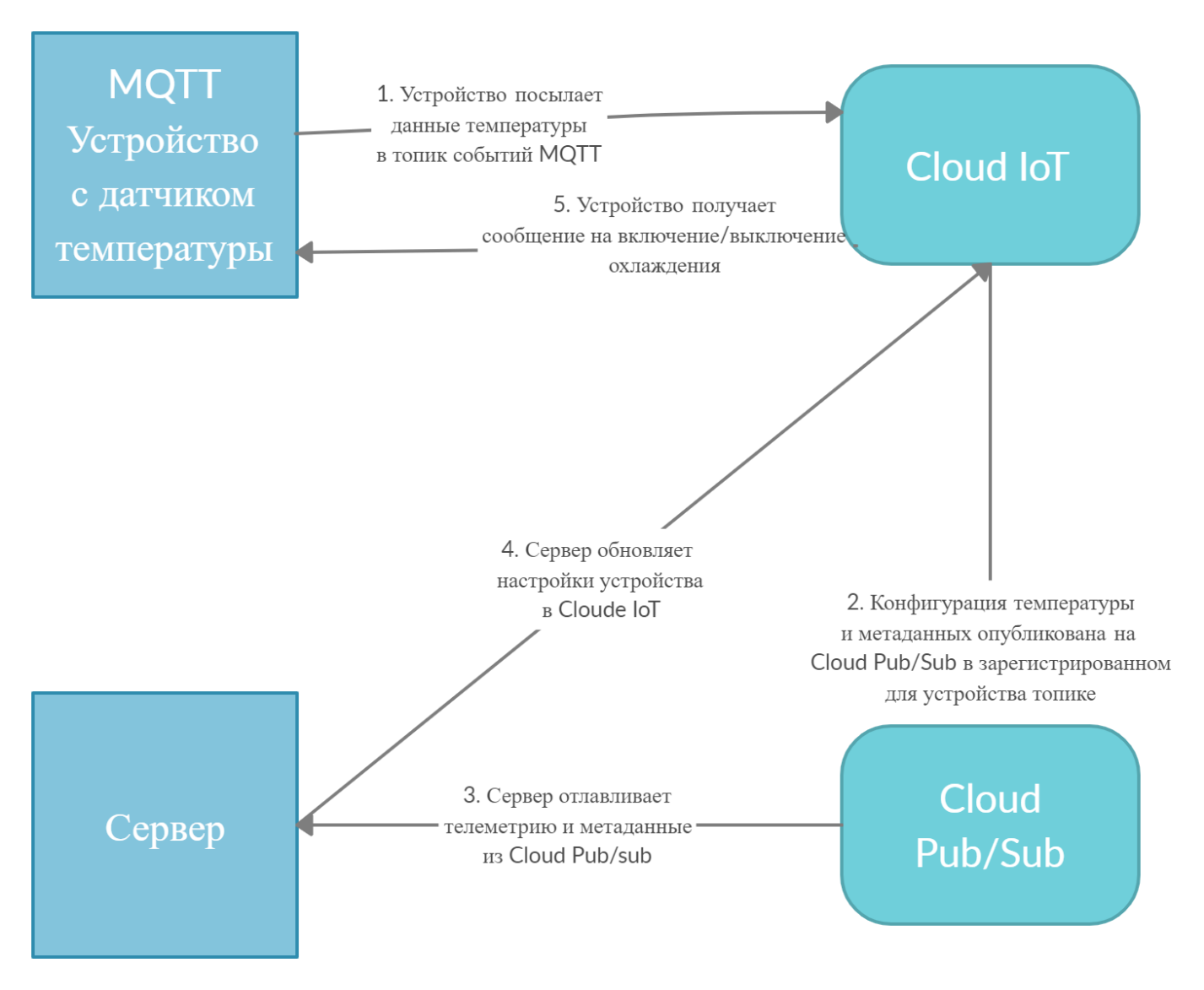
 Схематично, система, описанная выше, будет выглядеть следующим образом:

Рисунок 2.2.1 — схема приложения для платформы Google IoT Core

Поясним очередность действий:

1. Каждый температурный датчик устройства, подключенного в IoT систему, отправляет собранные телеметрические данные системе Cloud IoT для дальнейшей маршрутизации данных.
2. Система Cloud IoT размещает полученные данные каждого телеметрического устройства в Cloud Pub/Sub. Для каждого устройства применяется собственная, зарегистрированная ячейка событий.
3. Сервер занимается мониторингом ячеек событий в системе Cloud Pub/Sub и, как только в какой-то из них появляется новое событие, сервер отлавливает его. Далее следует анализ и обработка полученных сервером данных.
4. После того, как сервер получил результаты, он корректирует настройки элемента устройства, соответствующего температурному датчику, информацию от которого он отловил и обработал. Корректировка происходит путем отправки сообщения системе Cloud IoT.
5. Устройство получает сообщение от системы Cloud IoT с новыми настройками и корректирует их согласно сообщения.

В процессе проектирования будет выполнено создание реестра данной системы, а также создание устройства и добавление устройства в реестр. Так же будет выполнена подготовка устройства для передачи телеметрических данных. Рассмотрим настройку устройства, используя серверное управление, основанное на данных телеметрии.

Для начала работы нам понадобится новый проект на платформе Google Cloud Platform. Создадим его путем выбора кнопки «Создать проект» на главной странице. В предложенной форме дадим ему название, а в графе «Местоположение» выберем «Без организации». Выбираем «создать». После успешного создания проекта выберем его путем использования интерфейса верхней панели главной страницы, либо же откроем терминал по кнопке «Активизировать Cloud Shell». В открывшемся терминале пропишем следующую команду:

gcloud config set project <your-project-id>, где вместо <your-project-id> укажем идентификатор созданного проекта.

После того, как проект выбран, следует создать реестр Cloub Pub/Sub для данного проекта. Для начала создадим Pub-часть. Для этого в терминал введем следующую команду:

gcloud pubsub topics create tour-pub --project="${DEVSHELL\_PROJECT\_ID:-Cloud Shell}".

Данная команда создает Pub-часть в выбранном проекте, причем проект выбирается, путем использования встроенного механизма получения значения с применением зарезервированных имен. В данном случае это будет

"${DEVSHELL\_PROJECT\_ID:-Cloud Shell}"

Данная команда возвращает идентификационный номер выбранного проекта. Идентификационный номер проекта служит значением для ключа --project. Теперь займемся созданием Sub-части проекта. Для этого пропишем в консоли следующую команду:

gcloud pubsub subscriptions create tour-sub --topic=tour-pub.

Данная команда создает Sub-часть проекта, основываясь на уже созданной Pub-части. После ключа --topic следует имя созданной Pub-части.

После создания Cloud Pub/Sub создадим устройство, которое будет отправлять системе телеметрические данные и принимать новые настройки конфигурации. Так как любой разработчик заинтересован в безопасном обмене данными и сохранности получаемой и принимаемой информации от устройства и к устройству, нам следует сгенерировать публичный и приватный RSA ключи. Cloud IoT Core поддерживает RSA и алгоритм эллиптической кривой. В данной работе будет использован RSA алгоритмом. Данная криптосистема полностью закроет вопрос безопасности и с полной уверенностью сможем принимать за правду получаемую информацию. Cloud IoT Core использует ассиметричную аутентификацию:

* Устройство использует приватный(секретный) ключ для того, чтобы подписать JSON WEB-TOKET (JWT). Токен передается в Cloud IoT Core, как доказательство достоверности подключенного устройства.
* Система использует публичный ключ устройства, который загружен до того, как пришел JWT для проверки личности устройства.

Для того, чтобы автоматически сгенерировать 2048-битный приватный RSA ключ и само-заверяющий X.509 сертификат с SHA-256 подписью, введем следующую команду в Power Shell:

openssl req -x509 -newkey rsa:2048 -days 3650 -keyout rsa\_private.pem   
-nodes -out rsa\_public.pem -subj "/CN=unused".

Рассмотрим подробнее эту команду. Данная команда создает RSA ключ с само-заверяющим сертификатом X.509. По умолчанию срок действия сертификата X.509 истекает через 30 дней после создания. Чтобы установить количество дней до истечения срока действия сертификата, нами был добавлен флаг –days во время создания. Если будет произведена попытка создать или обновить устройство с истекшим сертификатом, или попытка подключить устройство к реестру, а срок действия сертификата реестра истек, Cloud IoT Core вернет ошибку. Как показано выше, сертификат, созданный описанной выше командой будет действовать 3650 дней. В данной команде используются инструменты командной строки openssl. OpenSSL является надежным, коммерческим и полнофункциональным набором инструментов для TLS (Transport Layout Security) и SSL (Secure Sockets Layer) протоколов. Это так же универсальная криптографическая библиотека.

После создания приватного и публичного RSA ключей с названиями rsa\_private.pem и rsa\_public.pem соответственно, создадим устройство в системе, которое будет использовать публичный ключ. Для создания устройства нам понадобится регион созданного нами Cloud IoT проекта. По умолчанию предлагается us-central-1. Так же нам понадобится имя созданного ранее Cloud IoT реестра. Нами был создан реестр temp-registry. Эти параметры являются обязательными. Команда по созданию устройства содержит так же множество других аргументов, но нам из них всех пригодится лишь аргумент, принимающий имя публичного ключа. Целиком команда выглядит следующим образом:

gcloud iot devices create (DEVICE : --region=REGION --registry=REGISTRY)  
[--auth-method=AUTH\_METHOD] [--blocked] [--device-type=DEVICE\_TYPE] [--log-level=LOG\_LEVEL] [--metadata=[KEY=VALUE,…]]  
[--metadata-from-file=[KEY=PATH,…]]  
[--public-key=[path=PATH,type=TYPE,[expiration-time=EXPIRATION-TIME],…]] [GCLOUD\_WIDE\_FLAG …].

В нашем исполнении команда будет выглядеть следующим образом:

gcloud iot devices create temp-dev --region=us-central1 \  
--registry=temp-registry \  
--public-key path=rsa\_public.pem,type=rs256.

temp-dev — название созданного нами устройства.

**ВЫВОДЫ**

В данном разделе нами была спроектирована Cloud IoT Core система, а также созданы Pub/Sub раздел, реестр и устройство. В следующей главе нами будет представлена реализация сервера, принимающего информацию из Pub/Sub и анализирующего ее, а также будет реализовано устройство, которое в зависимости от полученных данных будет регулировать настройки конфигурации. Демонстрация реализованного приложения будет происходить путем использования встроенного облачного терминала Power Shell.

ГЛАВА 3  
РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЙ ИНТЕРНЕТА ВЕЩЕЙ

3.1 Разработка приложения для облачного сервиса Microsoft Azure IoT.

Первым шагом разработки системы интернета вещей для сервиса, предоставляемого компанией Microsoft будет создание программного модуля, реализующего регистрацию, а если регистрация была до этого успешно проведена, то подключение устройства к центру IoT, с последующей выдачей ключа устройства для его дальнейшего использования в приложении. Создадим решение и добавим в него консольное приложение (.Net Framework). Название проекту можно дать любое, но хотелось бы понимать с чем имеешь дело, поэтому назованием проекта будет DeviceIdentity. В обозревателе решений перейдем в NuGet Package Manager и добавим данному приложению пакет Microsoft.Azure.Devices, предоставляемый компанией Microsoft, как API для программной работы с устройствами и использования его разработчиками.

Для начала, добавим в стандартный класс Program статический объект класса RegistryManager, входящий в пакет Microsoft.Azure.Devices. Данный класс используется для выполнения операций над устройствами, таких как создание, удаление и обновление. В данной работе будет использоваться объект данного класса для добавления/подключения устройства к сервису. В методе, являющейся точкой входа данного консольного приложения, первым делом определим данный объект. Есть два способа:

1. Конструктор по умолчанию.
2. Статический метод класса CreateFromConnectionString, принимающий единственный параметр: string connectionString.

Конструктор по умолчанию создает пустой объект, который в дальнейшем заполняется и изменяется методами объекта. Статический метод же возвращает объект, заполненный данными, соответствующими состоянию облачного сервиса на момент запроса. Строку подключения можно получить из IoT центра. Для этого зайдем во вкладку «Политики общего доступа». В данной вкладке находятся все политики(настройки), необходимые для корректной работы и взаимодействия с сервисом, устройствами и прочими модулями. Из предложенных вариантов выберем iothubowner. Данная политика будет использоваться нами для подключения объекта RegistryManager. Выберем первичный ключ строки подключения. Он и будет использоваться нами для получения состояния сервиса. Теперь все готово для создания объекта. Объявим статический объект и проинициализируем его.

registryManager = RegistryManager.CreateFromConnectionString(connectionString);

Следующим шагом будет создание метода, реализующего регистрацию/подключение устройства к центру. Объявим статический асинхронный метод типа Task и дадим ему название AddDeviceAsynk(). Первым делом объявим в методе переменную типа Device и назовем ее device. Данный тип так же входит в добавленный нами ранее пакет Microsoft.Azure.Devices. Данный класс в полной мере предоставляет все возможности, необходимые для качественной обработки и модифицирования устройства. Класс обладает 2 конструкторами (конструктор без параметров и конструктор, имеющий единственный параметр: string id), остальные возможности взаимодействия представлены в виде свойств. Среди свойств представлены следующие:

|  |  |
| --- | --- |
| Имя свойства | Описание |
| Authentication | Механизм аутенфикации устройства. |
| Capabilities | Возможности, доступные на устройстве. |
| CloudToDeviceMessageCount | Число сообщений, отправленных устройству из облака. |
| ConnectionState | Состояние устройства. |
| ConnectionStateUpdatedTime | Дата и время, когда устройство было в последний раз обновлено. |
| ETag | Etag устройства. |
| GenerationId | Идентификатор поколения устройства. |
| Id | Идентификатор устройства. |
| LastActivityTime | Дата и время, когда устройство было активно в последний раз. |
| Scope | Область, к которой принадлежит это устройство. |
| Status | Статус устройства. |
| StatusReason | Причина, по которой Устройство находится в указанном Статусе, если таковая имеется |
| StatusUpdatedTime | Дата и время последнего обновления статуса. |

Таблица 1

Далее, в методе создадим переменную типа string и назовем ее deviceId. Данная переменная будет хранить в себе уникальное имя для устройства. Запишем туда «roomTemperatureDevice». Название соответствует функциям, которые будет выполнять данное устройство. Следующим шагом будет создание блока try...catch, реализующего регистрацию, а если такое устройство уже было создано, то подключение к устройству сервиса. В блоке try проинициализируем переменную типа Device, созданную ранее результатом асинхронной функции AddDeviceAsync объекта класса RegistryManager. Данный метод принимает в качестве аргумента объект типа Device. Передадим туда новый объект с аргументом deviceId. Выглядеть это будет следующим образом:

device = await registryManager.AddDeviceAsync(new Device(deviceId));

Далее, при успешной регистрации, выведем на консоль сообщение о том, что данное устройство успешно добавлено. Но если устройство с таким уникальным идентификатором уже зарегистрировано в сервисе, то данный метод вызовет исключение типа DeviceAlreadyExistsException. Блок catch будет отлавливать исключения именно этого типа. С помощью данного блока будем обрабатывать ситуацию, когда такое устройство уже существует. Проинициализируем переменную типа Device, созданную ранее результатом асинхронной функции GetDeviceAsync объекта класса RegistryManager. Данный метод принимает в качестве аргумента переменную string deviceId. Выглядеть это будет следующим образом:

device = await registryManager.GetDeviceAsync(deviceId);

Далее, при успешном подключении, выведем на консоль сообщение о том, что данное устройство успешно подключено. После блока try…catch финальным шагом в разработке статической функции AddDeviceAsynс будет вывод на консоль ключ устройства, которое в итоге получили. Для получения ключа устройства воспользуемся свойством объекта Authentification. У данного свойства имеется свойство SymmetricKey, у которого есть PrimaryKey. Данный ключ будет использоваться нами в дальнейшем для создания клиента устройства. На этом разработка функции завершена.

В методе Main вызовем созданную выше функцию и удостоверимся, что все работает так, как нам требуется. В консоли, при успешной отработке, должно отобразиться следующее:

Device: "roomTemperature" connected.  
Device: "roomTemperature" Device key: "yy1iORYqx77aW0K7SqHY2X92mfp70VL9C\*\*\*\*\*\*\*\*\*\*=".

Сохраним ключ данного устройства, он пригодится нам далее.

Следующим шагом разработки системы интернета вещей для сервиса, предоставляемого компанией Microsoft будет создание программного модуля, реализующего клиент устройства для отправки сообщений, хранящих информацию о температуре и влажности, регистрируемые данным устройством в текущий момент времени с промежутком, установленным нами.

Первое, что нам потребуется, это создать в решении новое консольное приложение на .Net Framework. Назовем его согласно задаче, которую оно будет выполнять — «DeviceSimulation». В классе, в наличии которого имеется функция, являющейся входной точкой работы программы, объявим статическую переменную типа DeviceClient и назовем ее так же deviceClient. Объект данного класса предоставляет возможности по разносторонней работе с получением и отправкой сообщений в центр IoT. Далее понадобится информация о том, куда посылается сообщение, и от кого посылается сообщение. Куда посылается — uri центра IoT, а от кого посылается — ключ устройства, который сохранился после предыдущего запуска приложения по регистрации/подключению конкретного устройства. Заведем 2 статические переменные типа string и будем в них хранить нужную нам информацию.

Чтобы получить uri центра IoT зайдем в облачный сервис и перейдем во вкладку «Обзор». Перед нами появится страница, имеющая вкладку «Основное». Раскроем ее и найдем в появившихся пунктах «Имя узла». Скопируем содержимое данного пункта. Это будет uri цента IoT. Присвоим двум переменным соответствующие значения. Далее, в функции Main проинициализируем переменную класса DeviceClient, созданную ранее. Для инициализации воспользуемся статическим методом класса DeviceClient под названием Create. Класс определяет целых шесть методов с таким именем, но воспользуемся методом, принимающим в качестве параметров uri центра IoT, объект, реализующий интерфейс IAuthenticationMethod и значение enum TransportType.

Проговорим подробнее интерфейс IAuthenticationMethod. Он определяет единственный метод, возвращающий IotHubConnectionStringBuilder и принимающий единственным параметром объект типа IotHubConnectionStringBuilder, носящий название Populate. Данный метод заполняет экземпляр IotHubConnectionStringBuilder на основе свойств текущего экземпляра. Данный интерфейс служит для определения метода аутентификации устройства в центре IoT.

В качестве объекта, реализующего данный интерфейс будет использоваться объект класса DeviceAuthenticationWithRegistrySymmetricKey. Конструктор данного класса принимает два аргумента. Первым является идентификационный номер, в нашем случает строка «roomTemperature», а вторым ключ устройства, хранимый нами в одной из статических переменных исходного класса. В последний аргумент передадим тип протокола передачи информации mqtt. В итоге инициализация объекта клиента будет выглядеть следующим образом:

deviceClient = DeviceClient.Create(iotHubUri, new DeviceAuthenticationWithRegistrySymmetricKey("roomTemperature", deviceKey), TransportType.Mqtt);.

Следующим шагом разработки приложения по отправке сообщений в центр IoT от имени устройства будет создание асинхронного метода SendDeviceToCloudMessagesAsync() реализующего отправку информации с устройства о температуре и влажности воздуха в центр IoT. Так как разработка ведется лишь над симуляцией устройства и данное приложение является лишь демонстрационным, то будем симулировать значения температуры и влажности путем использования псевдо-генератора случайных чисел в заданном диапазоне. Объявим переменную типа double с названием minTemperature, которая будет отвечать за нижний порог температуры и проинициализируем ее значением равным двадцати. Объявим переменную типа double с названием minHumidity, которая будет отвечать за нижний порог влажности в комнате и проинициализируем ее значением равным 60.

Так же нам потребуется счетчик сообщений, который будет исполнять роль фабрики, генерирующей уникальные идентифицирующие номера сообщений, которые планируется отправлять. Уникальными они будут только в области локального запуска, но для отличия их от других и поочередного перехвата нам хватит и этого. В роли счетчика будет выступать переменная типа int с названием messageId, значение которой будет равняться единице.

Далее нам нужен генератор псевдослучайных чисел. В роли генератора возьмем объект встроенного класса Random. Объект данного класса обладает всеми необходимыми нам функциями. Объявим данный объект и проинициализируем при помощи пустого конструктора класса Random.

Теперь нужно реализовать часть метода, отвечающую за отправку информации центру IoT через заданный промежуток времени. Для этого объявим цикл while и в условии пропишем true. В теле цикла будут реализованы генерация, упаковка и отправка сообщения облачному сервису. Первым шагом будет генерация информации. Температуру будем генерировать путем добавления к минимальной температуре числа 15, умноженного на результат метода NextDouble() объекта класса Random. Данный метод возвращает значение, лежащее в промежутке от 0 до 1. Таким образом будем получать значение температуры из промежутка [20, 35]. Сохранять это значение будем в переменную типа double с названием currentTemperature. Аналогичным способом будет происходить получение влажности, но модифицироваться умножением на 20, а не на 15. В результате значение влажности будет лежать на промежутке [60, 80]. Сохранять значение влажности будем в переменную типа double с названием currentHumidity. Данные действия будут записаны следующим образом:

double currentTemperature = minTemperature + rand.NextDouble() \* 15;  
double currentHumidity = minHumidity + rand.NextDouble() \* 20;

Следующим шагом будет упаковка информации в сообщение. Для этого, в первую очередь, объявим анонимный объект с названием telemetryDataPoint, который будет выступать в роли содержимого сообщения, которое в дальнейшем будет собрано и отправлено облачному сервису. Содержимое сообщения будет включать идентификационный номер сообщения, идентификатор устройства, значение температуры и значение влажности. Этого нам должно хватить для отправки и предоставления информации. Проинициализируем созданный анонимный объект следующими свойствами:

* messageId = messageId++
* deviceId = "roomTemperature"
* temperature = currentTemperature
* humidity = currentHumidity

После создания анонимного объекта конвертируем его в формат JSON и сохраним полученную строку в переменную типа string с названием messageString. Следующим шагом упаковки будет создание объекта типа Message. Данный класс в полной мере предоставляет возможности по созданию сообщений, удовлетворяющих всем условиям пользователя. При создании объекта воспользуемся конструктором данного класса, принимающим единственным параметром массив значений типа byte. Чтобы из строки, получившейся в результате конвертирования анонимного объекта в JSON, получить массив значений типа byte, воспользуемся статическим методом GetBytes() класса Encoding, принимающего единственным параметром строку. Так же, для демонстрации возможностей объекта класса Message, добавим к существующим записям, еще одну, которая будет сигнализировать об опасности данной температуры для человека.

Для этого у объекта вызовем свойство с названием Properties, которое является объектом типа IDictionary<string, string>. У данного интерфейса есть метод Add, благодаря которому и будет происходить добавление новой записи в существующее сообщение. Метод Add, в нашем случае, будет принимать и в качестве ключа, и в качестве значения объект типа string. Передадим ключом туда строку «temperatureAlert», а значение ключа будет определять тернарное выражение, проверяющее текущую температуру, если она более 30 градусов, то в качестве значения пойдет true, иначе false. Данная запись будет выглядеть следующим образом:

message.Properties.Add("temperatureAlert", (currentTemperature > 30) ? "true" : "false");

Настало время реализовать последний шаг данного приложения — отправку сообщения на сервер. У нас имеется клиент, у нас имеется сообщение. Для отправки сообщения в центр IoT вызовем асинхронный метод объекта сообщения с названием SendEventAsync принимающим объект типа Message. Вызовем данный метод и передадим ему в аргументы созданное ранее сообщение. Добавим так же вывод в консоль того, что отправляется со стороны устройства и в каком часу это отправляется. Завершим метод тем, что приостановим поток на 5 секунд, чтобы при демонстрации работы сообщения не вываливались целой пачкой, либо же не ждать сообщения слишком долго. Осталось только вызвать данный метод в функции Main. Но запускать пока рано, ибо по подписке, используемой в данной работе, возможно посылать только 100 сообщений в сутки и нам не хотелось бы потратить их все, так и не перехватив ни одного.

Финальным шагом разработки системы интернета вещей для сервиса, предоставляемого компанией Microsoft будет создание программного модуля, реализующего перехват сообщений, приходящих в центр IoT по конкретному пути и вывод данных сообщений на консоль.

Создадим новый проект в решении и дадим ему название Connector. В классе, имеющим метод, являющийся точкой входа, объявим объект класса EventHubClient. Данный класс является основополагающим классом для получения и отправки событий центру IoT. Далее, при помощи этого класса будет происходить получение и обработка пришедших сообщений.

Проинициализируем объект данного класса, путем вызова статического метода CreateFromConnectionString. Класс EventHubClient имеет пять статических методов получения нового объекта типа EventHubClient. Выбранный нами метод принимает 2 аргумента типа string. Первым аргументом является строка подключения, которая была получена в приложении регистрации/авторизации устройства в центре IoT. Вторым аргументом является путь, из которого будем получать сообщения. Если с первым все понятно, то со вторым нужно пояснить. В предыдущем проекте использовали метод SendEventAsync для отправки сообщения, но из названия можно понять, что отправляли событие, носящее в себе информацию сообщения, которое передали в качестве аргумента метода. В Центре IoT события, которые были отправлены, можно найти по пути messages/events. Именно этот путь и будем использовать.

Для удобства вынесем две переменные с названиями connectionString и iotHubD2cEndpoint в поля класса и проинициализируем их соответствующими значениями, полученными ранее. В названии переменной, хранящей путь к событиям, которые хотим обработать есть аббревиатура D2C. Данная аббревиатура обозначает Device-To-Cloud, что с английского переводится как «от устройства к облаку». Она говорит о том, что данная переменная хранит именно путь к сообщениям от устройства.

Проинициализируем объект класса EventHubClient используя функцию, описанную ранее, с аргументами, строки подключения и пути к событиям. Так как данная программа создана для получения всех сообщений, приходящих в центр IoT, получаемых при помощи концентратора событий, то нам для начала нужно получить все возможные идентификаторы разделов, чтобы потом из каждого раздела отлавливать приходящие сообщения.

Для этого объявим переменную типа var с названием d2cPartitions, но на самом деле у нее будет тип string[], var использовали для сокращения записи. Созданную переменную проинициализируем путем вызова у созданного ранее объекта класса EventHubClient метода GetRuntimeInformation(). Данный метод извлекает информацию о среде выполнения концентраторов событий, которая требуется для создания объектов EventHubSender или EventHubReceiver. Возвращается объект класса EventHubRuntimeInformation. Данный объект содержит следующие свойства:

* CreatedAt — получает или задает время создания концентратора событий.
* PartitionCount — получает или задает количество разделов в концентраторе событий.
* PartitionIds — получает или задает массив идентификаторов разделов для концентратора событий.
* Path — получает или задает путь к концентратору событий.

На данном этапе нам понадобится свойство PartitionsIds, котрое и вызовем после вызова функции GetRuntimeInformation(). В результате получим следующую запись:

var d2cPartitions = eventHubClient.GetRuntimeInformation().PartitionIds;

Мы имеем объект концентратора событий, имеем идентификаторы разделов у данного концентратора событий, но нужно так же позаботиться об управляемом завершении каждого потока, занимающимся мониторингом сообщений в выделенном ему разделе. Для этого используем встроенный механизм CancellationTokenSource. Данный класс представляет собой фабрику токенов — объектов, получающих уведомление на завершение. Класс CancellationTokenSource содержит 3 конструктора:

* CancellationTokenSource() — конструктор по умолчанию.
* CancellationTokenSource(Int32) — конструктор, принимающий кол-во миллисекунд, через которое объект отправит всем своим токенам сигнал завершения.
* CancellationTokenSource(TimeSpan) — конструктор, принимающий объект классса TimeSpan, по завершении которого объект отправит всем своим токенам сигнал завершения.

Целью является завершать выполнение потоков тогда, когда этого захочет пользователь, поэтому воспользуемся конструктором по умолчанию:

CancellationTokenSource cts = new CancellationTokenSource();

Для получения токенов у данного объекта существует свойство Token, возвращающее дочерний, для данного источника токенов, объект класса CancellationToken. Теперь у нас имеется механизм корректного завершения.

Следующим шагом будет написание функции потока, отвечающей за мониторинг конкретного раздела концентратора событий. Объявим приватную статическую асинхронную функцию, возвращающую Task, с названием ReceiveMessagesFromDeviceAsync(). Так как данная потоковая функция должна знать о разделе, в котором ей нужно проводить мониторинг, первым ее аргументом будет объект типа string с названием partition. Так же нам требуется реализовать механизм безопасного завершения потоковой функции, для этого нам потребуется объект CancellationToken, который будет приниматься вторым параметром. В итоге заголовок функции будет выглядеть следующим образом:

private static async Task ReceiveMessagesFromDeviceAsync(string partition, CancellationToken ct).

Так как функция отвечает за мониторинг, то нам нужен объект, который в состоянии отслеживать изменения в концентраторе событий. Такой объект нам предоставляется классом EventHubReceiver. Данный класс представляет собой клиента, используемого в операциях приема событий из концентратора событий. Представляет логического получателя из определенного раздела. Предоставляет все необходимые методы для получения и отслеживания событий. Объект данного класса будем получать с использованием функций переменной eventHubClient.

Данный объект содержит метод GetDefaultConsumerGroup(), возвращающий объект класса EventHubConsumerGroup. Данный объект представляет собой группу потребителей по умолчанию для операции приема данных события. У данного объекта содержится метод CreateReceiver(). Данный метод принимает аргументами строку, являющуюся идентификатором раздела и время, с которого данный получатель начнет работать. Запись будет выглядеть следующим образом

var eventHubReceiver = eventHubClient.GetDefaultConsumerGroup().CreateReceiver(partition, DateTime.UtcNow);

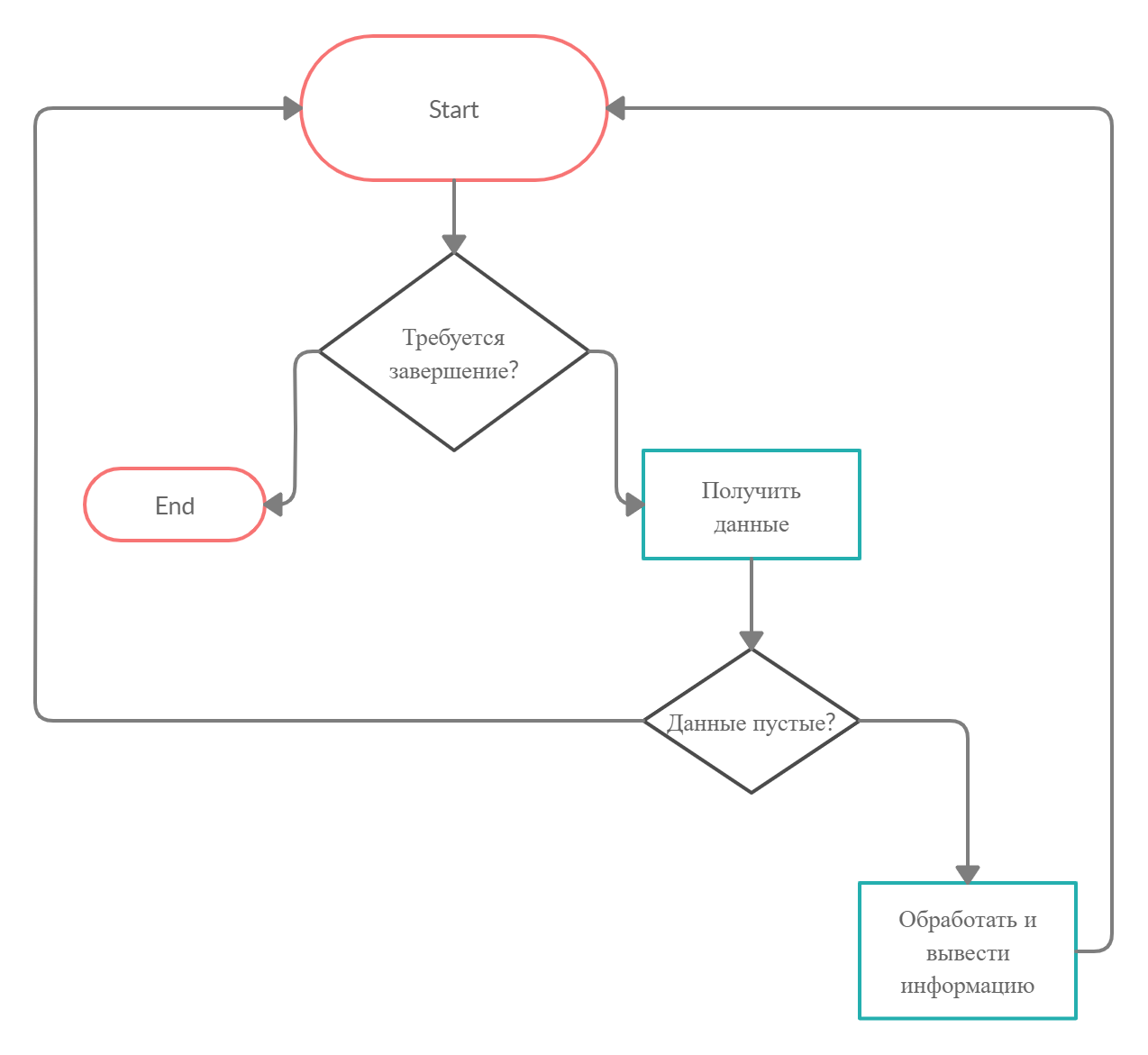
Теперь осталось реализовать механизм получения и обработки сообщений, полученных из раздела. Так как завершение работы будет происходить только по сигналу, то в худшем случае поток должен работать бесконечно долго. Для этого реализуем следующую схему:  
  


Рисунок 3.1.1 — схема механизма получения и обработки сообщений

Реализовывать данную схему будем в бесконечном цикле вида while(true){…}. В качестве проверки будет выступать блок if, условием которого будет сигнальное состояние завершения токена, полученного в качестве параметра. Если данное условие выполняется, то вызываем команду break; цикл завершается, функция оканчивает свою работу. Иначе пытаемся получить событие путем использования метода ReceiveAsync() объекта eventHubReceiver. Данный метод возвращает объект класса EventData.

Данный класс представляет событие, отправленное и полученное из потока концентратора событий. Он содержит тело события, набор определяемых пользователем свойств и различные метаданные, описывающие событие, такие как его смещение в разделе и его номер в последовательности потока. Разделы заполняются последовательностью данных о событиях.

Если объект, полученный функцией нулевой, то есть данные пустые, то вызываем команду continue и начинаем новую итерацию, иначе получаем в виде строки содержание события, путем использования метода GetBytes() объекта EventData и конвертирования полученного массива байт в строку, используя статический метод GetString(byte[] bytes) статического свойства UTF8, возвращающего Encoding статического абстрактного класса Encoding. Выглядеть данная запись будет следующим образом:

string data = Encoding.UTF8.GetString(eventData.GetBytes());

Выводим полученную строку в консоль и начинаем новую итерацию. Разработка потоковой функции мониторинга завершена.

Теперь разработаем механизм, позволяющий безопасно завершить работу каждого из заданий, занимающихся мониторингом новых событий, поступающих в центр IoT. Для этого реализуем обработку нажатия сочетания клавиш, которые чаще всего используются для завершения консольного приложения, а именно Ctrl + C. Для этого воспользуемся встроенным статическим классом System.Console.

Данный класс содержит статическое событие типа делегата ConsoleCancelEventHandler, носящее название CancelKeyPress. Данный делегат определяет метод, принимающий в качестве параметров object sender и ConsoleCancelEventArgs e.

В качестве метода, который привяжется к данному событию воспользуемся анонимный метод, представленный в виде лямбда-выражения. Данная лямбда будет принимать два параметра с названиями s и e, которые будут входными аргументами типа object и ConsoleCancelEventArgs соответственно.

System.Console.CancelKeyPress += (s, e) =>

Далее изменим свойство Cancel входного параметра e на значение true. После этого требуется сообщить всем заданиям о завершении работы, для этого у объекта класса CancellationTokenSource cts вызовем метод Cancel(). Выведем в консоль сообщение о том, что работа завершается

Console.WriteLine("Exiting...");

Весь механизм будет выглядеть следующим образом:

System.Console.CancelKeyPress += (s, e) => {  
 e.Cancel = true;  
 cts.Cancel();  
 Console.WriteLine("Exiting...");  
};

Финальным шагом разработки проекта будет создание массива/коллекции объектов типа Task, каждый из которых будет представлять задание — выполнение написанной ранее функции. Для этого объявим переменную var с названием tasks и проинициализируем ее объектом типа List<Task> путем использования конструктора данного класса:

var tasks = new List<Task>();

Далее, для каждого идентификатора раздела из полученного ранее множества идентификаторов создадим новое задание(поток выполнения с потоковой функцией), в параметры которой передадим сам идентификатор и новый токен, полученный источником токенов:

foreach (string partition in d2cPartitions) tasks.Add(ReceiveMessagesFromDeviceAsync(partition, cts.Token));

После того, как все задания(потоки) запущены, дождемся окончания выполнения каждого из заданий путем использования статической функции WaitAll(params Task[] tasks) класса Task. В качестве параметра функции предадим созданный ранее List<Task> tasks конвертировав его в массив:

Task.WaitAll(tasks.ToArray());

Наступил этап запуска приложений по отправке сообщений центру IoT и клиента обработки данных сообщений. Для этого нажмем правой кнопкой мыши на решение и выберем вкладку Properties. В появившемся окне перейдем в Common Properties > Startup Project.

Выберем радиокнопку Multiple startup projects и напротив названий Simulation Device и Connector выберем значение Start. Нажмем «Применить». Запускаем. После запуска появляются два консольных окна, одно из которых — процесс отправки, а другое — процесс приема. Выглядит это следующим образом:

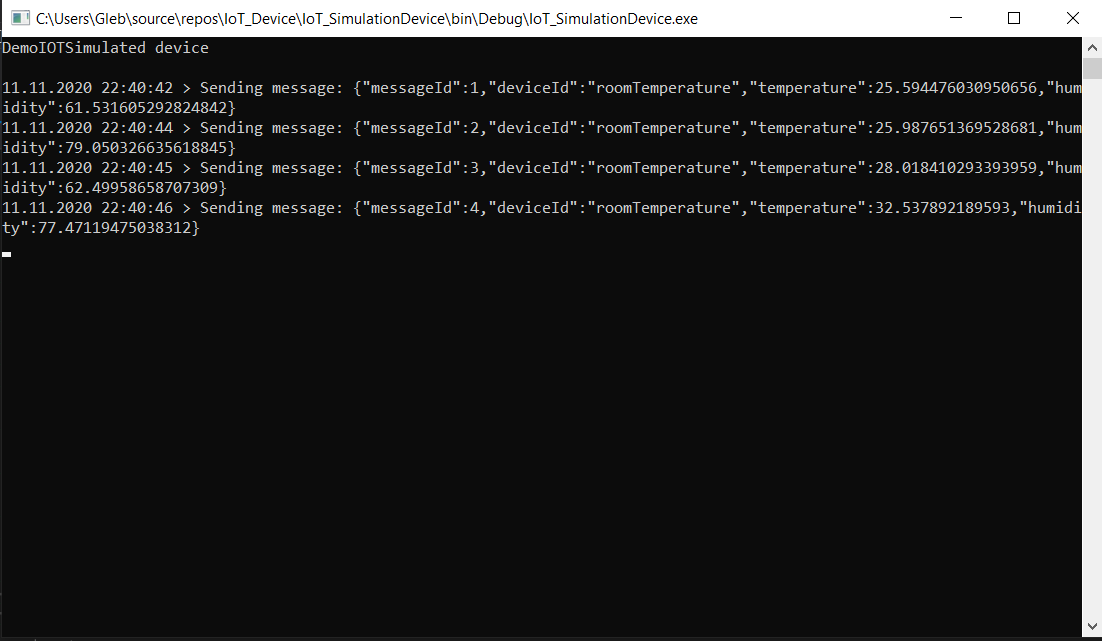


Рисунок 3.1.2 — результаты, отправленные центру IoT

На данном снимке экрана видна работа симулятора устройства, а именно дата, когда данное сообщение было отправлено и содержимое сообщения. Как видно, значение температуры не ниже 20 и не больше 35, аналогично влажность соблюдает выбранные нами рамки, так же видно, что сообщения отправляются приблизительно каждую секунду. Приблизительно, потому что уходит так же время на создание объектов, наполнение сообщения информацией, а так же отправку сообщения. Теперь приведем снимок экрана, на котором приводится информация, обработанная клиентом мониторинга сообщений:

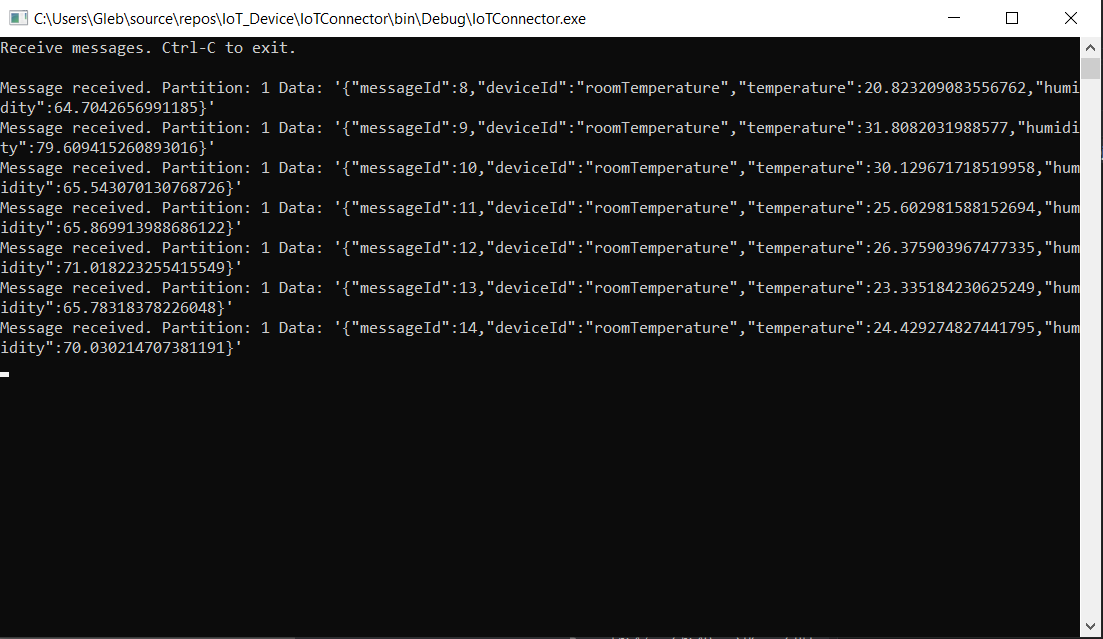


Рисунок 3.1.3 — данные, полученные клиентом

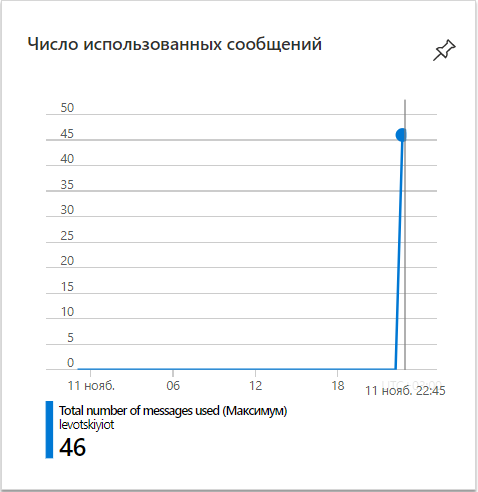
На данном снимке видна аналогичная по содержанию и контексту информация, что была отправлена из симулятора устройства, лишь за исключением отсутствия даты, когда клиент принял данное сообщение. Так же приведем снимок экрана, на котором изображен график, отражающий активность в центре IoT:

Рисунок 3.1.4 — график использованных сообщений

На данном графике видно, что симулятор устройства отправил 46 из 100 доступных в день сообщений. В Центре IoT приводятся так же другие графики активности, но они приводиться не будут.

На этом закончена разработка решения включающая в себя совокупности приложений, реализующих регистрацию/авторизацию устройства, симуляцию устройства на примере термометра/датчика влажности воздуха, а также клиента многопоточного мониторинга и обработки событий, поступающих центру IoT. Данные приложения могут стать базой для построения более сложных проектов и систем, но они демонстрируют основы разработки системы интернета вещей.

3.2 Разработка приложения для облачного сервиса Google IoT Core

Первым шагом разработки системы интернета вещей для сервиса, предоставляемого компанией Google будет создание программного модуля, реализующего устройство, включающее в себя датчик температуры, а также охлаждающее устройство. Работа данного программного модуля будет представлять из себя отправку сообщений Google Cloud Pub/Sub системе, содержащих телеметрические данные данного устройства, а также прием и обработку сообщений, включающих в себя настройки, корректирующие работу охлаждающей системы. Результатом разработки будет являться модуль, демонстрирующий работу устройства, а также реализующий взаимодействие с облачным сервисом, а именно отправка и обработка пришедших сообщений. Разработка будет происходить на языке Python 3.0.

Для большей гибкости, при использовании готового модуля, зададим несколько аргументов, которые обязательны для модуля при запуске. Список всех аргументов приведен ниже:

* --project\_id — данный аргумент содержит идентификационное имя проекта, в котором находится данное устройство.
* --registry\_id — данный аргумент содержит идентификационное имя регистра проекта, к которому происходит подключение.
* --device\_id — данный аргумент содержит идентификационный номер устройства, которое данный программный модуль будет представлять.
* --private\_key\_file — данный аргумент содержит путь к файлу, которой содержит приватный ключ для подключения и идентификации устройства.
* --algorithm — данный аргумент содержит название алгоритма шифрования, использовавшегося для генерации JWT.

Данные аргументы являются базовыми и обязательны для запуска данного программного модуля. Приведем пример запуска программного модуля из консоли с использованием данных аргументов:  
$ python telemetry\_device.py \

--project\_id=some-project-id \

--registry\_id=some-registry-id \

--device\_id=some-device-id \

--private\_key\_file=rsa\_private.pem \

--algorithm=RS256

Так же будем распознавать необязательные аргументы. Данные аргументы будут содержать значения по умолчанию. Приведем список данных аргументов:

* --cloud\_region — данный аргумент содержит в себе название региона, в котором зарегистрирован проект. Значением по умолчанию является строка “us-central1”. Так как созданный ранее проект расположен именно в us-central1, то его можно смело опускать при запуске.
* --ca\_certs — данный аргумент содержит в себе название корневого сертификата. Значением по умолчанию является строка “roots.pem”. Данный сертификат будет использоваться для проверки однорангового узла.
* --num\_messages — данный аргумент содержит в себе число сообщений, которое будет отправлено Cloud Pub/Sub системе. Значением по умолчанию является 100. Данного количества нам должно вполне хватить для успешной демонстрации работы охлаждающего устройства.
* --mqtt\_bridge\_hostname — данный аргумент содержит в себе наименование хоста, при помощи которого будет производиться взаимодействие между устройством и облаком.
* --mqtt\_bridge\_port– данный аргумент содержит в себе наименование порта, на который будут отправляться сообщения и с которого будут считываться сообщения.
* --message\_type — данный аргумент описывает тип сообщений, которые будут отправляться в Cloud Pub/Sub систему.

Первым шагом разработки программного модуля будет написание метода-парсера входных аргументов. Данный метода будет возвращать объект, содержащий в себе все аргументы. Для этого воспользуемся и импортируем библиотеку argparse. Метод будет носить название parse\_command\_line\_args. Данный метод будет состоять из этапов объявление переменной парсера, совмещенное с его инициализацией, добавление парсеру новых аргументов, содержащих описание, а так же значение по умолчанию, и возврат результата, полученного после обработки входных аргументов, а именно объекта аргументов командной строки.

После объявления метода, объявим объект, с говорящим именем parser и проинициализируем его при помощи конструктора argparse.ArgumentParser(). Данный конструктор содержит аргумент, принимающий описание данного парсера. После успешного объявления и инициализации добавим все аргументы, необходимые для правильной работы данного программного модуля. Добавление происходит путем вызова метода add\_argument() объекта парсера.

Данный метод содержит несколько реализаций, нам же понадобится неполный набор из того, что предлагает данный метод. В приведенном ниже примере демонстрируется добавление нового аргумента, охватывающего все, необходимые нам свойства:

parser.add\_argument(  
'--cloud\_region',  
 default='us-central1',  
 help='GCP cloud region').

Первым аргументом метода добавления нового аргумента является имя, под которым будет находиться интересующая нас информация. Следующим аргументом является значение по умолчанию и задается ключевым словом default. Последним, но не менее важным, является аргумент, принимающий строку помощи, описывающую, что конкретно принимает данный аргумент и зачем он нужен. После корректного добавления всех аргументов и соответствующих им свойств, вернем результат обработки командной строки, вызвав return parser.parse\_args(). Код всего метода приведен в приложении А.

Описанный метод будет вызываться первым в функции main, являющейся стартовой точкой программного модуля. В данной функции вызовем описанный выше метод и присвоим его результат объекту с названием args:

args = parse\_command\_line\_args().

После того, как стартовые данные успешно получены, настал момент создания MQTT клиента, который будет взаимодействовать с Cloud Pub/Sub системой. Для создания данного клиента воспользуемся библиотекой paho.mqtt.client и импортируем ее под псевдонимом mqtt. Данная библиотека содержит конструктор для создания нужного нам клиента. Конструктор носит имя Client. Данный конструктор принимает аргумент client\_id. Данный аргумент имеет строковое представление и имеет следующую структуру:

“projects/{}/locations/{}/registries/{}/devices/{}”,

где вместо фигурных скобок должны быть идентификационное имя проекта, наименование локации проекта, наименование регистра, а также идентификационный номер устройства соответственно.

Проинициализируем объект client данным конструктором и передадим ему в качестве аргумента следующие значения:

client = mqtt.Client(  
 client\_id='projects/{}/locations/{}/registries/{}/devices/{}'.format(  
 args.project\_id,  
 args.cloud\_region,  
 args.registry\_id,  
 args.device\_id)).

Как представлено выше, тремя из пяти обязательных аргументов запуска модуля программа уже воспользовалась. Далее, присвоим объекту клиента значения имени и пароля для аутенфикации при помощи метода username\_pw\_set() объекта client. Данный метод принимает параметр username и password. В качестве параметра установим значение ‘unused’, а вот полю password присвоим объект JWT. Для того, чтобы получить готовый JWT, потребуется написать метод, возвращающий такой токен.

Объявим метод create\_jwt. Данный метод будет служить сборщиком JWT исходя из полученных аргументов. В качестве аргументов будем передавать данному методу идентификационное имя проекта, полученное из аргументов запуска, путь к файлу с приватным ключом, а также название алгоритма шифрования, использовавшегося для генерации данного приватного ключа. Для начала сгенерируем анонимный объект с названием token. Данный объект будет иметь три поля: ‘iat’, ‘exp’ и ‘aud’.

Данные поля обозначают время когда токен был создан, время, когда токен станет не валидным, а также массив чувствительных к регистру строк или URI, являющийся списком получателей данного токена. Когда принимающая сторона получает JWT с данным ключом, она должна проверить наличие себя в получателях — иначе проигнорировать токен, соответственно. Далее, после того, как границы валидности и получатели описаны, следует получить содержимое файла, лежащего на пути, указанном в качестве пути к файлу с приватным ключом для данного устройства. Для это отроем файл на чтение и считаем всю необходимую информацию. Для этого воспользуемся следующими методами:

with open(private\_key\_file, 'r') as f:  
 private\_key = f.read()

Теперь все готово для того, чтобы получить готовый JWT. Для этого импортируем класс jwt. Данный класс содержит метод encode. Данный метод принимает на вход параметры полезной нагрузки токена, приватный ключ, а также наименование алгоритма. Все вышеперечисленные значения у нас имеются, так что выполнить данный метод не составит труда. Результатом выполнения и будет JWT, необходимый для аутенфикации пользователя. Целиком метод будет выглядеть следующим образом:

def create\_jwt(project\_id, private\_key\_file, algorithm):  
 token = { 'iat': datetime.datetime.utcnow(),  
 'exp': datetime.datetime.utcnow() + datetime.timedelta(minutes=60),  
 'aud': project\_id }  
 with open(private\_key\_file, 'r') as f:  
 private\_key = f.read()  
 return jwt.encode(token, private\_key, algorithm=algorithm).

В данном случае создается токен, состояние валидности которого сохраняется в течение 60 минут. Это не верно со стороны гибкости кода, но для исследовательской демонстрации нам этого должно вполне хватить.

После реализации метода присвоим в качестве аргумента password метода username\_pw\_set объекта client результат метода create\_jwt. В качестве аргументов передадим args.project\_id, args.private\_key\_file и args.algorithm соответственно. Далее следует настроить параметры сетевого шифрования и аутенфикации для клиента, при поддержке SSL/TLS. Для этого воспользуемся методом tls\_set() объекта client. Данный метод принимает в качестве входных аргументов:

* ca\_certs — данный аргумент представляет строковый путь к файлам сертификатов центра сертификации, которые должны рассматриваться как доверенные этим клиентом. Если это единственный вариант, клиент будет работать аналогично веб-браузеру. Другими словами, брокеру потребуется сертификат, подписанный центрами сертификации в ca\_certs, и он будет взаимодействовать с использованием TLS v1, но не будет пытаться выполнить аутентификацию в какой-либо форме. Это обеспечивает базовое сетевое шифрование, но может быть недостаточным в зависимости от того, как настроен брокер. По умолчанию в Python 2.7.9+ или 3.4+ используется центр сертификации системы по умолчанию. В более старых версиях Python этот параметр является обязательным.
* certfile, keyfile — данные аргументы представляют строки, указывающие на сертификат клиента в кодировке PEM и закрытые ключи соответственно. Если эти аргументы не равны None, они будут использоваться в качестве информации о клиенте для аутентификации на основе TLS. Поддержка этой функции зависит от брокера. Обратим внимание на то, что если любой из этих файлов зашифрован и для его расшифровки требуется пароль, Python запросит пароль в командной строке. В настоящее время невозможно определить обратный вызов для предоставления пароля.
* cert\_reqs — данный аргумент определяет требования к сертификату, которые клиент предъявляет к брокеру. По умолчанию это ssl.CERT\_REQUIRED, что означает, что брокер должен предоставить сертификат.
* tls\_version — данный аргумент указывает версию используемого протокола SSL / TLS. По умолчанию (если версия python поддерживает это) определяется самая высокая версия TLS. Если же данная возможность недоступна, используется TLS v1. Предыдущие версии (все версии, начинающиеся с SSL) возможны, но не рекомендуются из-за возможных проблем с безопасностью.
* ciphers — данный аргумент представляет собой строку, определяющую, какие шифры шифрования допустимы для этого соединения, или None, чтобы использовать значения по умолчанию.

Вызовем данный метод и передадим ему в качестве аргументов ca\_certs и tls\_version значения args.ca\_certs и ssl.PROTOCOL\_TLSv1\_2, соответственно.

На данный момент клиент сконфигурирован лишь со стороны идентификации и аутенфикации, а также описаны методы шифрования для безопасного взаимодействия с облаком. Следующим этапом будет создание класса, реализующего работку клиента, в том числе обработку полученных сообщений, отправку собственных сообщений, а также работу некоторых других событий. Для данных целей нами будет создан класс с говорящим названием Device. Данный класс будет в полной мере описывать необходимые нам методы. Начнем с реализации конструктора. Конструктор данного класса не будет иметь входных аргументов. Он всегда будет определять внутри себя необходимые для работы поля. В качестве таких полей будут выступать поле для хранения значения температура, поле, для хранения состояния охлаждающего устройства (вкл(true)/выкл(false)), а также поле состояния подключения (true/false). Ниже приведена реализация данного конструктора по умолчания класса Device:

def \_\_init\_\_(self):  
 self.temperature = 0  
 self.fan\_on = False  
 self.connected = False

Изначально, все состояния имеют значения false, а значение температуры равняется нулю. Далее, нам предстоит реализовать функции, срабатывающие на клиентские события объекта client:

* on\_connect — данная callback функция вызывается, когда брокер отвечает на запрос подключения. Метод обязан иметь следующие аргументы:
  1. client — экземпляр клиента для обратного вызова.
  2. userdata — личные данные пользователя, установленные в конструкторе Client() или user\_data\_set().
  3. flags — флаги ответа, отправленные брокером.
  4. rc — результат соединения.
* on\_publish — данная callback функция вызывается, когда сообщение, которое должно было быть отправлено с помощью вызова publish(), завершило передачу брокеру. Для сообщений с уровнями QoS 1 и 2 это означает, что соответствующие взаимоподключения выполнены. Для QoS 0 это просто означает, что сообщение покинуло клиент. Переменная mid соответствует переменной, возвращаемой соответствующим вызовом publish(), что позволяет отслеживать исходящие сообщения.
* on\_disconnect — данная callback функция вызывается, когда клиент отключается от брокера. Имеет аналогичные аргументы, что и функция on\_connect, за исключением отсутствия аргумента flags.
* on\_subscribe — данная callback функция вызывается, когда брокер отвечает на запрос подписки. Переменная mid соответствует переменной, возвращаемой соответствующим вызовом subscribe (). Переменная grant\_qos — это список целых чисел, которые определяют уровень QoS, предоставленный брокером для каждого из различных запросов подписки.
* on\_message — данная callback функция вызывается, когда было получено сообщение по теме, на которую подписывается клиент, и сообщение не соответствует существующему обратному вызову фильтра темы. Для определения обратного вызова, который будет вызываться для определенных фильтров темы, используется message\_callback\_add(). on\_message будет служить запасным вариантом, если не найдено ни одного совпадения. Использует аргументы client и usetdata, описанные выше в методе on\_connect, а также аргумент message. Данный аргумент является экземпляром MQTTMessage. Этот класс содержит участников, тему, полезную нагрузку, и многое другое.

Начнем с метода, реализующего обработку подключения, а именно on\_connect. В теле класса Device объявим метод с именем on\_connect. Данный метод будет выводить в консоль сообщение о том, что подключение произошло успешно, при этом информация о подключении будет получаться путем вызова метода error\_str, реализация которого будет приведена после реализации всех callback функций. Все, что следует знать сейчас, это то, что данный метод принимает аргумент rc из callback функции.

Далее установим поле connected объекта класса в активное положение, а именно присвоим ему значение true. Аналогичным образом реализуем callback функцию on\_disconnect класса Device, только, разумеется, значение connected установим в неактивное состояние, присвоим ему значение false. Далее реализуем callback функцию on\_publish. В реализации данной функции нам необходимо только вывести отладочную информацию о том, что сообщение опубликовано. Следовательно, выведем в консоль сообщение ‘Published message acked’.

Следующей реализацией будет метод on\_message. Данный метод будет вызываться, когда устройство перехватит сообщение по подписке. Присвоим переменной payload полезную нагрузку сообщения, декодированную в формат ‘utf-8', так как изначально это просто множество нечитаемых байт. Теперь нам следует вывести содержимое сообщения, а именно полезную нагрузку, топик сообщения, а также качество обслуживания.

Данные параметры определяются значениями payload, message.topic и str(message.qos) соответственно. Устройство получит последнюю конфигурацию при подписке на тему конфигурации. Если для устройства нет конфигурации, устройство получит конфигурацию с пустой полезной нагрузкой. Поэтому обязательно следует проверить полезную нагрузку. Если таковая пуста, то следует завершить выполнение метода путем вызова оператора return.

Конфигурация передается в полезной нагрузке сообщения. В нашей реализации сервер отправляет сериализованную строку JSON. Получим ее, путем вызова метода load класса json и передачи ему в качестве аргумента переменную payload. Данные, полученные в JSON виде обязательно имеют ключ fan\_on. Если значение, полученное из сообщения и нынешнее значение объекта не совпадают, нам следует изменить состояние поля объекта, путем присвоения ему значения, полученного из сообщения, так же выведем в консоль текущее состояние объекта.

Нам осталось описать реализацию последнего метода, который будет выступать в роли callback функции, а именно on\_subscribe. В данном методе будем выводить качество обслуживания, а также проверим нулевой элемент качества обслуживания. Если значение данного элемента сравнимо с кодом 128, обозначающим, что подписка не удалась, то выведем это сообщение в консоль. Все методы переопределены. Определение методов находится в приложении Б.

Теперь нам следует указать клиенту, какие методы будут выступать в роли callback функций. Для этого у объекта client поочередно будем вызывать одноименные указатели на функции и присваивать им функции объекта device класса Device:

client.on\_connect = device.on\_connect  
client.on\_publish = device.on\_publish  
client.on\_disconnect = device.on\_disconnect  
client.on\_subscribe = device.on\_subscribe  
client.on\_message = device.on\_message

Клиент полностью настроен и готов к подключению к Cloud Pub/Sub системе. Для этого у объекта client вызовем метод connect(). Передадим данному методу, в качестве аргументов, хост и номер порта. Эти данные у нас хранятся в аргументах строки подключения и носят имена args.mqtt\_bridge\_hostname и args.mqtt\_bridge\_port соответсвенно.

После подключения нам следует активировать функцию-петлю. Что это такое и зачем нам оно потребуется? Mqtt клиент устроен таким образом, что у него есть 2 буфера сообщений, а именно на вход и на выход. Так как писать функции, которые работают с этими буферами и вызывают соответствующие методы, в зависимости от сообщения, трудоемко, существует встроенная функция loop(). Данная функция обрабатывает сообщения и вызывает соответствующие callback функции. Структура обработки сообщений mqtt клиента проиллюстрирована ниже:

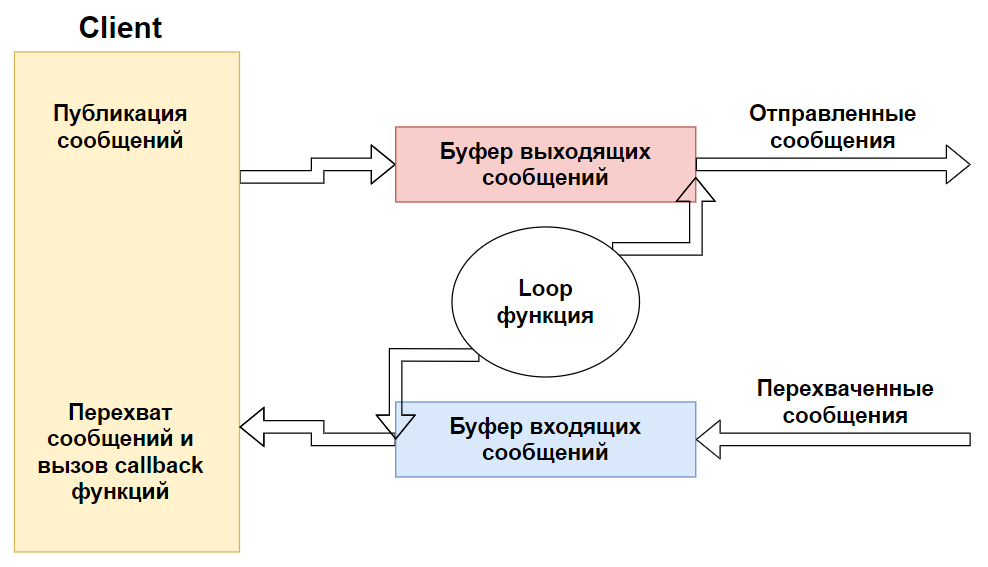


Рисунок 3.2.1 — клиентская схема обработки сообщений

Для того, чтобы начать перехватывать и обрабатывать сообщения, требуется вызвать метод loop\_start() объекта client. Loop\_start () запускает новый поток, который для вас через регулярные промежутки времени вызывает метод цикла. Он также автоматически выполняет повторное подключение. Чтобы остановить цикл, воспользуемся методом loop\_stop (). Внутри данного, если можно так можно сказать, блока, будем изменять температуру и посылать ее значение Cloud Pub/Sub системе. Измерение температуры будет в зависимости от состояния охлаждающего устройства.

Начнем с того, что подпишемся на топик конфигурации, предназначенный для конкретного устройства. Для этого нам потребуется строка/путь для конкретного топика конфигурации. Для каждого устройства он задается следующей маской: /devices/{device\_id}/config.

Все готово для подписки, но перед этим, подождем около 5 секунд, для того, чтобы дать время клиенту успешно подключиться к сокету. После ожидания вызовем метод subscribe объекта client. Данный метод в параметры принимает путь к топику конфигурации и код качества обслуживания. Передадим методу созданную выше строку и код качества обслуживания, равный единице. После вызова данного метода клиент подписался.

Следующий логический блок будет работать следующим образом: в цикле, размерностью args.num\_messages, значение по умолчанию которого равняется 100, на каждой итерации будет происходить обновление температуры, получаемой с симулируемого датчика, затем отправка сообщения Cloud Pub/Sub системе, содержащего информацию о текущей температуре, далее будет задержка в 1 секунду, конец итерации.

Обновление температуры будет происходить путем вызова метода update\_sensor\_data объекта Device, реализацию которого будет приведена далее. Данный метод не принимает аргументов. Реализация данного метода предельно простая. В зависимости от того, включено ли охлаждение или выключено, от значения температуры объекта Device будет отниматься 1 пункт, либо прибавляться 1 пункт.

После вызова данного метода следует отправить сообщение в систему. Для этого заведем переменную payload и присвоим ей результат функции dumps класса json. Данный метод принимает строку и преобразует ее вид в json. Далее отправим сгенерированное сообщение системе. Отправка будет происходить путем вызова метода publish объекта client.

Описанный выше метод в качестве параметров принимает путь к топику, где следует разместить это сообщение, само сообщение, а также коэффициент качества обслуживания. Второй и третий параметры у нас уже готовы, а первый параметр строится по аналогии с построением пути для подписки. Первому аргументу будет передана следующая строка:  
mqtt\_config\_topic = '/devices/{}/config'.format(args.device\_id).

Сообщение отправлено, осталось только подождать 1 секунду. Для этого вызовем метод time.sleep(1). После работы цикла следует отсоединиться вызвав метод disconnect() объекта client. Затем завершить работу loop функции, вызвав метод client.loop\_stop(). Работа клиентской части на этом закончена.

Если же нет необходимости ограничиваться каким-то размером сообщений, а целью является иметь бесконечно работающий клиент, следует заменить loop\_start() на loop\_forever().Листинг функции main() приведен в приложении В.

Теперь опишем создание сервера, который передает конфигурацию на устройства Google Cloud IoT. Далее будет представлен сервер, который использует данные телеметрии от нескольких устройств Cloud IoT. Устройства сообщают данные телеметрии, которые потребляет сервер из темы Cloud Pub/Sub системы. Затем сервер решает, включить или выключить индивидуальные устройства охлаждения. Как и клиент, при запуске, сервер нуждается в начальных данных. Начальные данные будут представлены в виде строки подключения. Ниже представлен пример запуска сервера из терминала:

$ python server.py \  
 --project\_id=project-id \  
 --pubsub\_subscription=topic-subscription \

Как и в клиенте, обработкой входных аргументов будет заниматься отдельным метод с названием parse\_command\_line\_args. Так как структура и все нетривиальные моменты по созданию данного метода были приведены в описании клиента, ниже приведена реализация данного метода для данного сервера:

def parse\_command\_line\_args():  
 parser = argparse.ArgumentParser(  
 description='Example of Google Cloud IoT registry and '  
 'device management.')   
 parser.add\_argument(  
 '--project\_id',  
 default=os.environ.get("GOOGLE\_CLOUD\_PROJECT"),  
 required=True,  
 help='GCP cloud project name.')  
 parser.add\_argument(  
 '--pubsub\_subscription',  
 required=True,  
 help='Google Cloud Pub/Sub subscription name.')  
  
 parser.add\_argument(  
 '--service\_account\_json',  
 default=os.environ.get("GOOGLE\_APPLICATION\_CREDENTIALS"),  
 help='Path to service account json file.')  
  
 return parser.parse\_args()

Далее займемся созданием класса Server, реализующего необходимую нам логику. Конструктор данного класса принимает аргумент service\_account\_json. Суть данного аргумента хранит свойство help в методе parse\_command\_line\_args. Данный аргумент используется для авторизации.

Для этого ипортируем класс compute\_engine из google.auth. У данного класса вызовем конструктор Credentials() и присвоим объект переменной credentials. Если данная переменная пустая, сообщаем в консоль об ошибке и завершаем работу. Далее создадим google api client. Данный клиент будет выступать в роли сервиса сервера. Импортируем класс discovery из googleapiclient. Но, перед тем, как создать клиент, подготовим все необходимое для этого. Нам понадобится discovery url, по которому будет происходит открытие сервиса. Его составим из DISCOVERY\_API и API\_VETSION. Данные макросы определены в скрипте и носят следующие значения:

API\_VERSION = 'v1'  
DISCOVERY\_API = 'https://cloudiot.googleapis.com/$discovery/rest'

А строка будет выглядеть следующим образом:  
discovery\_url = '{}?version={}'.format(DISCOVERY\_API, API\_VERSION)

После того, как все готово, присвоим полю \_service объекта класса Server значение, вернувшееся от метода build класса discovery. Данный метод принимает в качестве аргументов имя сервиса, версию API, discovery url, полномочия, полученные ранее, а также флаг кеширования. Имя сервиса будем хранить в макросе следующего вида:

SERVICE\_NAME = 'cloudiot'

Так же, для синхронизации работы с несколькими устройствами нам потребуется критическая секция. Целиком реализация конструктора находится ниже:

def \_\_init\_\_(self, service\_account\_json):   
 from google.auth import compute\_engine  
 credentials = compute\_engine.Credentials()  
 if not credentials:  
 sys.exit('Could not load service account credential '  
 'from {}'.format(service\_account\_json))  
  
 discovery\_url = '{}?version={}'.format(DISCOVERY\_API, API\_VERSION)  
  
 self.\_service = discovery.build(  
 SERVICE\_NAME,  
 API\_VERSION,  
 discoveryServiceUrl=discovery\_url,  
 credentials=credentials,  
 cache\_discovery=False)  
 self.\_update\_config\_mutex = Lock().

Следующим методом будет \_update\_device\_config. Данный метод будет реализовывать отправку новых настроек конфигурации конкретному устройству. Метод принимает в качестве входных аргументов идентификационное имя проекта, название региона, идентификационное имя регистра, идентификатор устройства, которому будет отправляться новые настройки конфигурации, а так же информация, полученная от устройства.

Объявим переменную confid\_data и присвоим ей значение None. Данная переменная будет хранить в себе настройки конфигурации, которые необходимо передать устройству. Далее следуют операции сравнения температуры с критическими значениями. Если значение температуры отрицательное, то нам следует выключить охлаждающее устройство устройства, если значение температуры больше десяти, то наоборот, следует включить охлаждающее устройство.

Следовательно, в переменную настроек конфигурации будет записано {'fan\_on': False}, либо {'fan\_on': True} соответственно. В любом другом случае изменение конфигурации настроек не требуется и можно смело завершать работу метода вызвав оператор return. После заполнения переменной новыми настройками, следует их отправить устройству. Так как устройство получает полезную нагрузку в виде json, то и отправлять следует строку вида json. Для этого проинициализируем новообъявленную переменную следующим образом:

config\_data\_json = json.dumps(config\_data)

Так же нам понадобится тело, сообщения, которое будет содержать версию обновления и бинарные данные. В конфигурации устройства указывается версия для обновления, которая используется, чтобы избежать гонки обновлений конфигурации. В этом случае воспользуемся специальным значением 0, которое сообщает Cloud IoT о необходимости всегда обновлять конфигурацию. Проинициализируем переменную body следующим образом:

body = {  
 'version\_to\_update': 0,  
 'binary\_data': base64.b64encode(  
 сonfig\_data\_json.encode('utf-8')).decode('ascii')  
}

Теперь нам остается только создать сообщение и отправить его. Созданное сообщение запишем в переменную request. Сообщение будем генерировать путем вызова иерархической цепочки методов созданного ранее сервиса:

request = self.\_service.projects().locations().registries().devices()  
.modifyCloudToDeviceConfig(name=device\_name, body=body),

где device\_name — строка, хранящая полный путь к конкретному устройству. Далее нам осталось только отправить сообщение.

Так как следующая операция требует синхронизации, откроем критическую секцию объекта, вызвав self.\_update\_config\_mutex.acquire(). Метод отправки сообщения так же может сгенерировать исключение типа HttpError, так что следует обрамить его вызов блоками try…catch…finaly. В блоке try вызовем request.execute(), что является отправкой сообщения, а в блоке catch будем отлавливать исключение и выводить его в консоль. Блок finaly будет отвечать за закрытие критической секции. На этом написание метода изменение настроек конфигурации отдельного устройства закончено.

До этого были описаны вспомогательные функции. Теперь будет описана главная функция данного класса, которая будет использовать описанные ранее методы. Данная функция называется run и будет отвечать за прослушивание подписки в Cloud Pub/Sub системе. Данная функция будет принимать аргументами идентификатор проекта, а также Pub/Sub подписку. Для того, чтобы появилась возможность получать и обрабатывать сообщения, нужно получить объект-подписчик.

Для этого импортируем класс pubsub из google.cloud. У импортированного класс pubsub вызовем SubscriberClient(). Результатом вызова проинициализируем переменную subscriber. Далее, получим путь подписки, воспользуясь методом subscrtption\_path объекта subscriber. В вызванный метод передадим идентификатор проекта, а также Pub/Sub подписку. Осталось только подписать объект subscriber по subscription\_path для получения и обработки сообщений. Обработкой сообщений должен заниматься callback метод. Опишем данный метод в теле метода run.

Данный метод будет носить не оригинальное, но свойственное выполняемой работе имя — callback. Данный метод будет принимать единственный параметр — message. Получив сообщение, попробуем распарсить его в json объект. Данную часть реализации внесем в блок try…except. В try части будем парсить, а в except части будем отлавливать ошибки типа ValueError и выводить полученную ошибку в консоль.

Если же считывание произошло успешно, получим из сообщения идентификатор проекта, идентификатор регистра, идентификатор устройства, с которого пришло сообщение, а также регион. Полученные данные передадим в метода update\_device\_config. На этом реализация callback метода окончена. Осталось вызывать у subscriber метод subscribe и передать туда subscription\_path и callback. Подписчик запущен и готов обрабатывать сообщения.

Теперь все готово для запуска сервера. В главной функции main вызовем метод parse\_command\_line\_args() и результат вызова передадим в переменную args. Вызовем конструктор класса Server, передав соответствующие параметры из переменной args. Вызовем метод run() объекта server и передадим в качестве параметров args.project\_id и args.pubsub\_subscription. Сервер запущен и готов к работе.

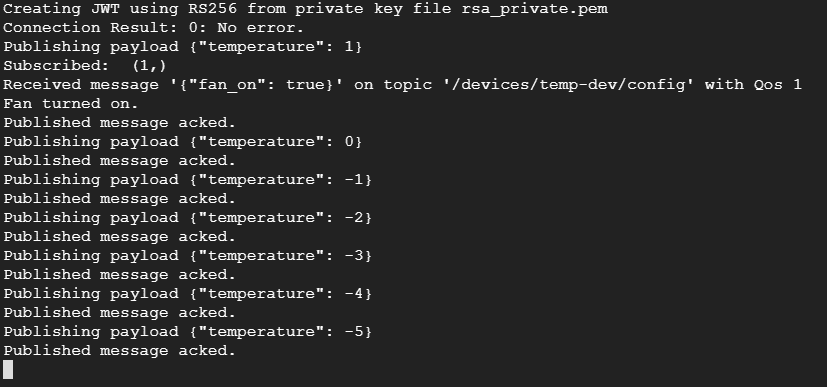
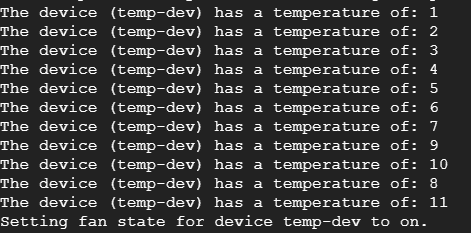
 В завершение исследования в рамках данной главы запустим в разных консолях сервер и, для демонстрации, одного клиента.

Рисунок 3.2.2 — Работа клиента.

Отработка сервера:

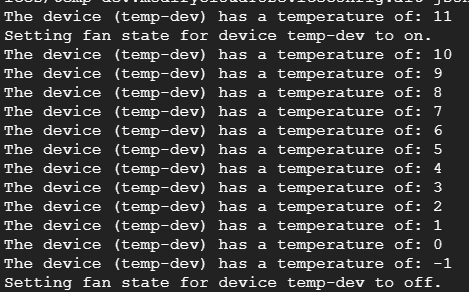
Рисунок 3.2.3 — Результат работы сервера

Рисунок 3.2.4 — Результат работы сервера

Графическое представление активности в облаке:

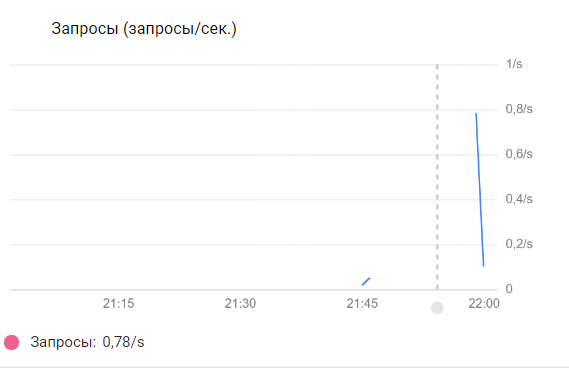


Рисунок 3.2.5 — график активности в облаке

**ВЫВОДЫ**

В данной главе были разработаны приложения под платформы интернета вещей корпораций Google и Microsoft с использованием предлагаемых компаниями библиотек для правильного и безопасного взаимодействия. Были рассмотрены модули, отвечающие за симуляцию устройств и модули, отвечающие за чтение сообщений, отображаемых на сервере платформы. Так же была рассмотрена разработка сервера, способного корректировать настройки каждого из устройств в отдельности. С использованием разработанных компонентов были построены системы контроля температурно-влажностного баланса, которые, в тандеме с реальными датчиками, можно запустить у себя дома.

Заключение

Подводя итоги, отметим, что в ходе проделанной работы были изучены основные аспекты и положения разработки облачно-ориентированных систем для платформ Microsoft Azure IoT Suite и Google Cloud Core.

Продемонстрировано проектирование схожих, но все же разных IoT систем. Система, разработанная для платформы Google IoT Core, продемонстрировала не только отправку и перехват сообщений, но и логику работы небольшого сервера по регуляции температурного режима всех устройств, подключенных к системе. Так же были продемонстрированы подготовительные работы по настройке и отладке проектов на стороне облака.

Описаны системы безопасности и их особенности при регистрации и подключении устройств. Описаны основные возможности мониторинга проделанной на облаке работы. Продемонстрированы графики, отражающие проделанную работу. Отражены все особенности создания репозиториев для конкретной задачи и рассмотрены аспекты маршрутизации для конкретного устройства.

Данная работа показывает, что при наличии технической составляющей, как минимум физических датчиков, способных передавать данные на сервер, моем создать приложение, отражающее информационную составляющую физических датчиков из любой точки мира.

На основе таких модулей, а также разработке более совершенных и сложных серверных модулей разработчик вполне способен создать целые управляемые системы на подобие умного дома и прочих современных систем.

В качестве сервера датчиков может выступать одноплатный компьютер из семейства Raspberry или схожих. А в качестве датчиков, как бы это странно не звучало, выступают физические датчики, подключенные к одноплатному компьютеру.

В будущей курсовой работе планируется развитие теме разработки IoT системы на основе Google IoT Core с использованием физических датчиков и физической модели сбора и обработки информации на основе Raspberry pi 4 model B. В систему планируется включить телеметрический датчик, датчик угарного газа, а также датчик движения. Так же планируется проанализировать еще одну платформу для разработки приложений с использованием «интернета вещей».

Современный мир не стоит на месте и с каждым днем выходи все больше и больше возможностей и платформ построение IoT систем. Появляется все больше возможностей сбора и обработки информации, анализа и изменения конфигурации устройств, позволяющих строить облачные системы быстрее, надежнее и, самое главное, качественнее. В скором времени нас будет окружать одна общая глобальная сеть, а получение той или иной информации будет занимать гораздо меньше времени, чем сейчас. Это заметно уже сейчас, так как имеется возможность наблюдать множество примеров совмещенных облачных систем от мировых компаний и производителей. Системы могут включать в себя множество блоков, каждый из которых захватывает ту или иную сферу жизни пользователя. Поэтому, многие аналитики оценивают данное направление, как перспективное, и в недалеком будущем оно будет одним из популярнейших направлений разработки в сфере IT.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. A Tour of Cloud IoT Core [Electronic resource]. — Mode of access: https://codelabs.developers.google.com/codelabs/cloud-iot-core-overview?hl=en#0. — Date of access: 01.12.2020.

2. Review of Google IoT Platform [Electronic resource]. — Mode of access: https://habr.com/ru/post/442710/. — Date of access: 01.12.2020.

3. Creating A Simulated Device For IoT Hub Using .NET [Electronic resource]. — Mode of access: https://www.c-sharpcorner.com/article/simulated-devices-for-iot-on-our-pc-using-net/ — Date of access: 01.12.2020.

4. Getting Started with Cloud IoT Core [Electronic resource]. — Mode of access: https://www.leverege.com/blogpost/getting-started-with-cloud-iot-core. — Date of access: 01.12.2020.

5. Azure IoT [Electronic resource]. — Mode of access: https://azure.microsoft.com/ru-ru/overview/iot/. — Date of access: 01.12.2020.

6. Google-IoT-Simple-Simulated-Device IoT [Electronic resource]. — Mode of access: <https://github.com/SamueleMeta/Google-IoT-Simple-Simulated-Device>. — Date of access: 01.12.2020.

Приложения

Приложение А

def parse\_command\_line\_args():  
 parser = argparse.ArgumentParser(  
 description='Example Google Cloud IoT MQTT device connection code.')  
 parser.add\_argument(  
 '--project\_id',  
 default=os.environ.get("GOOGLE\_CLOUD\_PROJECT"),  
 required=True,  
 help='GCP cloud project name.')  
 parser.add\_argument(  
 '--registry\_id', required=True, help='Cloud IoT registry id')  
 parser.add\_argument(  
 '--device\_id',  
 required=True,  
 help='Cloud IoT device id')  
 parser.add\_argument(  
 '--private\_key\_file', required=True, help='Path to private key file.')  
 parser.add\_argument(  
 '--algorithm',  
 choices=('RS256', 'ES256'),  
 required=True,  
 help='Which encryption algorithm to use to generate the JWT.')  
 parser.add\_argument(  
 '--cloud\_region', default='us-central1', help='GCP cloud region')  
 parser.add\_argument(  
 '--ca\_certs',  
 default='roots.pem',  
 help='CA root certificate. Get from https://pki.google.com/roots.pem')  
 parser.add\_argument(  
 '--num\_messages',  
 type=int,  
 default=100,  
 help='Number of messages to publish.')  
 parser.add\_argument(  
 '--mqtt\_bridge\_hostname',  
 default='mqtt.googleapis.com',  
 help='MQTT bridge hostname.')  
 parser.add\_argument(  
 '--mqtt\_bridge\_port', type=int, default=8883, help='MQTT bridge port.')  
 parser.add\_argument(  
 '--message\_type', choices=('event', 'state'),  
 default='event',  
 help=('Indicates whether the message to be published is a '  
 'telemetry event or a device state message.'))  
 return parser.parse\_args()

Приложение Б

def on\_connect(self, unused\_client, unused\_userdata, unused\_flags, rc):  
 print('Connection Result:', error\_str(rc))  
 self.connected = True  
  
 def on\_disconnect(self, unused\_client, unused\_userdata, rc):  
 print('Disconnected:', error\_str(rc))  
 self.connected = False  
  
 def on\_publish(self, unused\_client, unused\_userdata, unused\_mid):  
 print('Published message acked.')  
  
 def on\_subscribe(self, unused\_client, unused\_userdata, unused\_mid,  
 granted\_qos):  
 print('Subscribed: ', granted\_qos)  
 if granted\_qos[0] == 128:  
 print('Subscription failed.')  
  
 def on\_message(self, unused\_client, unused\_userdata, message):  
 payload = message.payload.decode('utf-8')  
 print('Received message \'{}\' on topic \'{}\' with Qos {}'.format(  
 payload, message.topic, str(message.qos)))  
 if not payload:  
 return  
 data = json.loads(payload)  
 if data['fan\_on'] != self.fan\_on:  
 self.fan\_on = data['fan\_on']  
 if self.fan\_on:  
 print('Fan turned on.')  
 else:  
 print('Fan turned off.')

Приложение В

def main():  
 args = parse\_command\_line\_args()  
 client = mqtt.Client(  
 client\_id='projects/{}/locations/{}/registries/{}/devices/{}'.format(  
 args.project\_id,  
 args.cloud\_region,  
 args.registry\_id,  
 args.device\_id))  
 client.username\_pw\_set(  
 username='unused',  
 password=create\_jwt(  
 args.project\_id,  
 args.private\_key\_file,  
 args.algorithm))  
 client.tls\_set(ca\_certs=args.ca\_certs, tls\_version=ssl.PROTOCOL\_TLSv1\_2)  
 device = Device()  
 client.on\_connect = device.on\_connect  
 client.on\_publish = device.on\_publish  
 client.on\_disconnect = device.on\_disconnect  
 client.on\_subscribe = device.on\_subscribe  
 client.on\_message = device.on\_message  
 client.connect(args.mqtt\_bridge\_hostname, args.mqtt\_bridge\_port)  
 client.loop\_start()  
 mqtt\_telemetry\_topic = '/devices/{}/events'.format(args.device\_id)  
 mqtt\_config\_topic = '/devices/{}/config'.format(args.device\_id)  
 device.wait\_for\_connection(5)   
 client.subscribe(mqtt\_config\_topic, qos=1)   
 for \_ in range(args.num\_messages):   
 device.update\_sensor\_data()  
 payload = json.dumps({'temperature': device.temperature})  
 print('Publishing payload', payload)  
 client.publish(mqtt\_telemetry\_topic, payload, qos=1)  
 time.sleep(1)  
 client.disconnect()  
 client.loop\_stop()  
 print('Finished loop successfully. Goodbye!')