Remote management framework for IoT devices utilizing Blockchain

A dissertation submitted for the Degree of Master of Computer Science

R.S. Deshapriya
University of Colombo School of Computing
2017

Abstract

In recent times, the Internet of Things has become a major driving force in the development of a connected world. Potentially millions of devices featuring sensors and actuators would form a complete, cohesive network of things that would generate billions of bits of data. There are many issues related to the management and security of IoT devices, posing risks and difficulties in authentication, and remote management of devices belonging to a single owner.

Blockchain has become popular with the rise of cryptocurrencies as a distributed, peer-to-peer ledger. With its features of verification, security and immutability, it has also become highly useful in areas other than cryptocurrency as a secure storage mechanism which is immune to attacks. As such, the possibility of implementing private blockchains which fulfill use cases related to secure storage, nonrepudiation and maintenance of reliable records has been explored.

This research was targeted at bringing the security of blockchain as a viable means to manage and maintain details of IoT devices including their credentials, as well as the enablement of remote management of multiple devices belonging to a single owner. The implementation focuses on two main flows: the creation and management of devices and the storage of messages on blockchain.

There are three main components of the developed system, the backend application which consists of a generic MQTT message broker and a RESTful web service API, the blockchain component which is implemented using Hyperledger Fabric, and an AngularJS web application which provides a standard interface for the user. The backend application impelements chaincode to interface with the blockchain to form smart contracts on the creation of device details and device message details on the ledger in the form of transactions.

Evaluation of the system was carried out through the implementation of Behaviour-Driven Development-based testing methodologies, and have conclusively proven that blockchain can be utilized as a secure storage mechanism for maintaining IoT device details.

Acknowledgements

There are many individuals who contributed greatly towards the completion of this research.

I would like to offer my heartfelt thanks to,

Dr. Kasun de Zoysa, whose mentorship and guidance has been a great boon, and whose knowledge has steered this work in the right direction,

my dear wife and family, whose patience and encouragement have always spurred me on,

and my friends and colleagues who have provided valuable insights and better solutions to the problems I sought to solve.

Contents

List of Tables

List of Figures

List of abbreviations

IoT	Internet of Things. 6–8, 10–17, 25
IP	Internet Protocol. 11
IPFS	InterPlanetary File System. 14
MQTT	Message Queue Telemetry Transport. 11, 12, 16, 17, 23
PBFT	Practical Byzantine Fault Tolerance. 15, 20

Chapter 1

Introduction

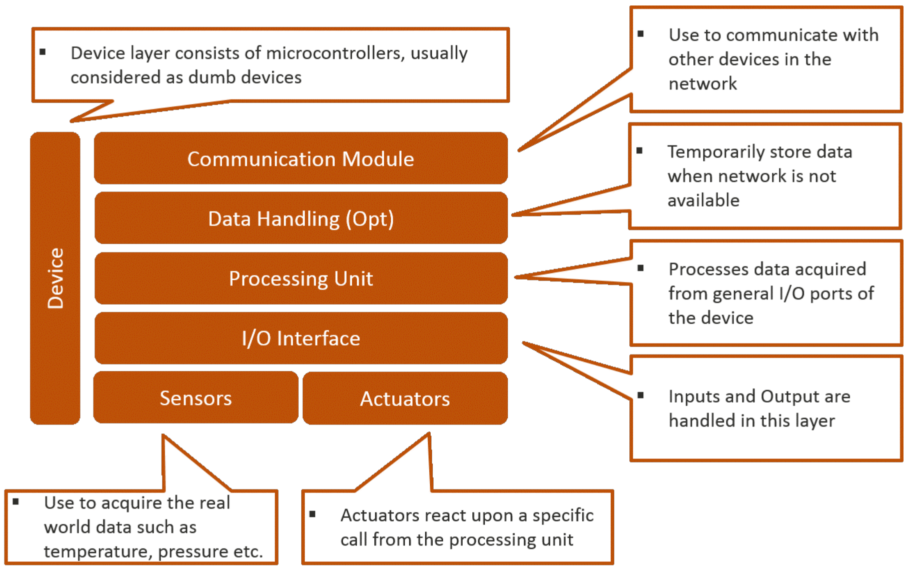
1.1 Overview

The Internet of Things is a conceptual vision of how all items in the modern world will connect to the Internet to send data via sensors and render received commands via actuators. IoT has grown in leaps and bounds in the recent past, with millions of devices being added to the Internet on a weekly basis. Beyond devices simply sending data in a limited environment over a small local network, IoT provides a framework for device data to be aggregated and distributed all around the world, thereby allowing remote management of devices, data aggregation and extraction, as well as analytics to be carried out on device data.

A typical IoT architecture contains the following components described in the sub-sections below.

1.1.1 Device Component

The device component contains the sensors and actuators that make up a device as well as communication modules to be able to send/receive messages from the Internet through a gateway or directly, as seen in Figure 1.1.



Device layer consists of microcontrollers, usually considered as dumb devices
Device
Communication Module
Data Handling (Opt)
Processing Unit
I/O Interface
Sensors
Actuators
Use to communicate with other devices in the network
Temporarily store data when network is not available
Processes data acquired from general I/O ports of the device
Inputs and Output are handled in this layer
Use to acquire the real world data such as temperature, pressure etc.
Actuators react upon a specific call from the processing unit

Figure 1.1: The device component of the IoT reference architecture

1.1.2 Gateway component

The device components are aggregated and managed by the gateway component, which in turn provides connectivity to the Internet over standard IoT communication protocols, as seen in Figure 1.2

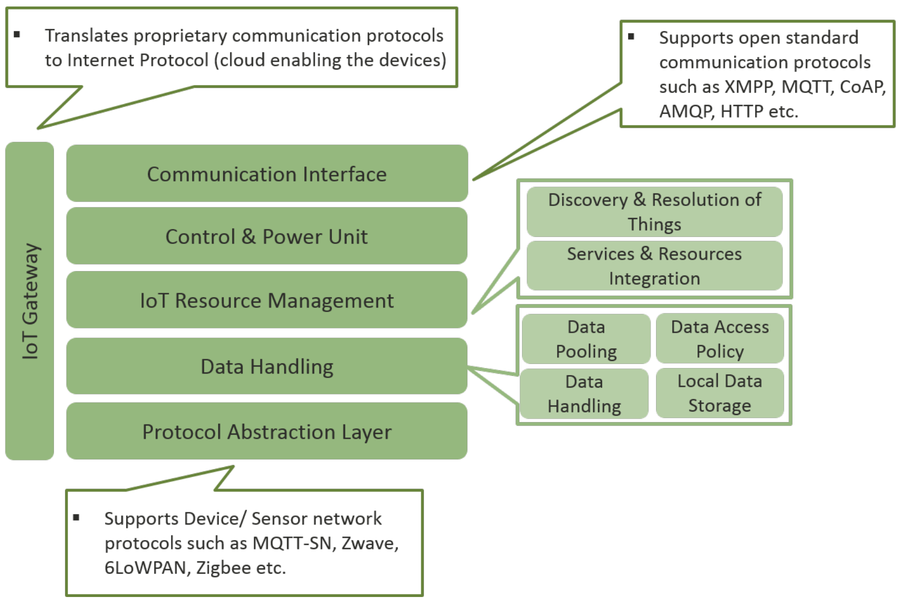

Translates proprietary communication protocols to Internet Protocol (cloud enabling the devices)
Supports open standard communication protocols such as XMPP, MQTT, CoAP, AMQP, HTTP etc.
IoT Gateway
Communication Interface
Control & Power Unit
IoT Resource Management
Data Handling
Protocol Abstraction Layer
Discovery & Resolution of Things
Services & Resources Integration
Data Pooling
Data Access Policy
Data Handling
Local Data Storage
Supports Device/ Sensor network protocols such as MQTT-SN, Zwave, 6LoWPAN, Zigbee etc.

Figure 1.2: The gateway component of the IoT reference architecture

Furthermore, the gateway acts as a conduit between low-level sensor and actuator devices which cannot muster the power or complexity needed to communicate with a cloud-based IoT broker.

1.1.3 Broker/Cloud component

The broker component lies on the cloud and handles aspects of device management as well as publishing and subscribing to messages sent by or to IoT gateways and devices. It also ties in to any other applications related to data analytics, data processing, or web service APIs that would make use of the data sent by IoT devices. This is illustrated in detail in Figure 1.3.
Although the above infrastructure and architectural patterns exist, there is no mechanism for IoT gateways to be considered as an abstract connectivity tools and for the underlying sensors and actuators, which are the actual IoT devices, to be considered individually and brought under the purview of a single owner user regardless of which gateway they are connecting through. The goal of this project is to provide this abstraction, and to make all devices deployed by a single user controllable using a single interface.

1.2 Problem description

IoT gateways are used to enable connectivity for base-level IoT sensor devices and to send their data to a hosted IoT message broker after enrichment, edge processing and formatting in to an accepted format. However, there is no mechanism to manage devices connecting to multiple IoT gateways which are under the same user in a unified manner. A user who makes use of multiple IoT devices connecting to multiple gateways must manage all gateways separately and not as part of a single cohesive network. IoT gateways belonging to the same organization/user have to be managed separately at the device manager level, over-complicating the management

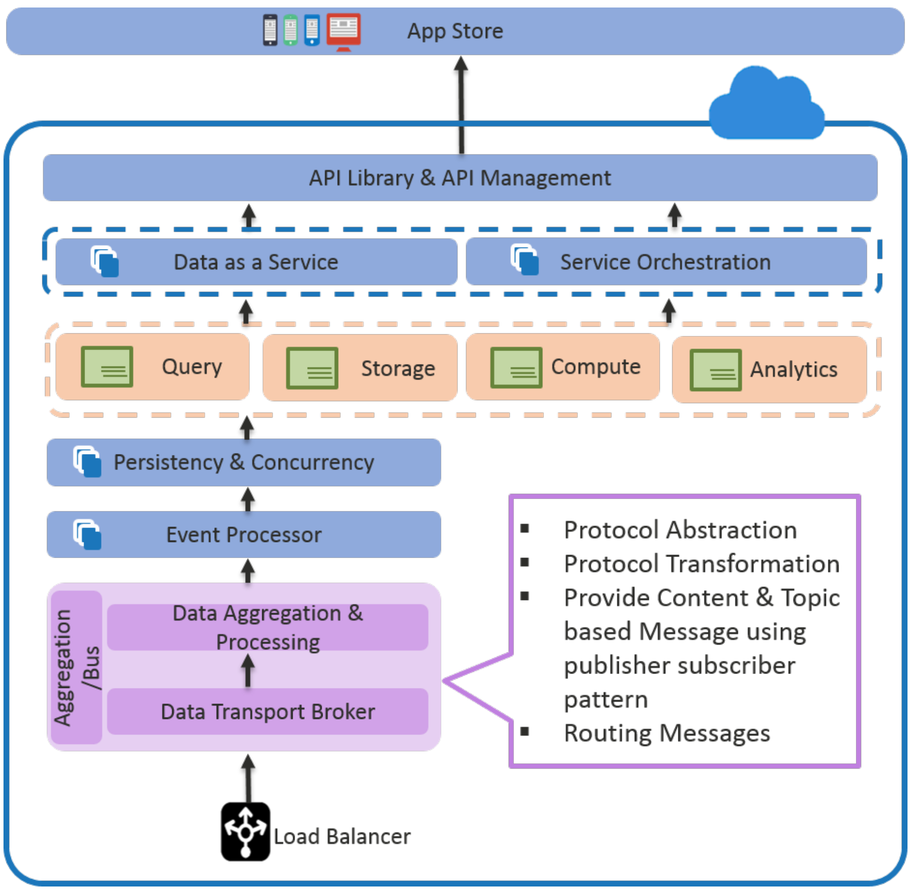


Figure 1.3: The cloud and service broker component of the IoT reference architecture

of sensor/actuator devices even further. As base-level sensor devices are the main concern for a user, it is cumbersome to have to manage gateways and not the devices themselves at the message broker/device management level. Furthermore, IoT networks are plagued with many security issues due to individual device credentials being compromised.

1.3 Goals and Objectives

The goal of this project is to integrate with a typical IoT broker framework to ensure that IoT devices (sensors and actuators) connecting to the gateway component of the framework can be controlled regardless of physical gateway they are connecting through, and to provide secure storage for all device details and messages on a private Blockchain.
A typical IoT broker framework is designed to act as a gateway framework consolidating device messages through their management, message brokering and maintenance of messaging topics. This proejct will utilize a typical IoT framework's message brokering capabilities and provide a management and message storage framework which provides the necessary levels of security. Blockchain, the secure ledger for cryptocurrency applications, will be used to store message details and credentials providing an extra layer of security for sensitive device data.

- Build a generic IoT message broker framework which sends and receives messages.
- Design and build a connectivity layer which consolidates messages based on the connecting gateway and owner of the gateway.

- Design and build a device management web service layer which provides secure storage for device details and credentials on the Blockchain.
- Analyse literature on IoT gateway and broker architecture and identify best approach and practices for implementation of the project.
- Implement a device management dashboard that allows users to manage all their devices connecting through multiple gateways from a single user interface

1.4 Scope of project

The following items fall within the scope of the project:

- Design and development of generic IoT message broker framework
- Design and development of connectivity layer which manages abstraction of message broker connectivity and providers remote management of IoT devices
- Design and development of a device management layer which utilizes Blockchain as its secure storage

The following assumptions have been made:

- All implementations of Blockchain provide a generic interface for interaction with other components
- The IoT devices considered for the project will all require a gateway through which then connects to an IoT broker, and are not edge processing devices that directly connect to the IoT broker

1.5 Contents of dissertation

This dissertation contains a literature review, where the various sources of research and literature related to the subject area are analyzed and discussed, followed by an analysis and design section, where the design of the proposed solution is discussed along with the salient points related to the methodology used. This is followed by a section on the implementation of the solution, where the various implementation details and technologies used as well as the integration points between the different implemented components are discussed. The next section is on the evaluation methodology of the solution, where the testing methodology used to evaluate the solution is explained and the test results are presented. This is followed by a conclusion, analyzing the built solution and its viability of use.

Chapter 2

Literature review

This literature review is the result of a comprehensive study of research undertaken in the field of IoT with regard to gateways and connectivity components. Literature was analyzed and tabulated in categories related to the following outlined sections.

2.1 IoT Device-to-gateway communication

IoT devices and gateways communicate in a number of different ways, using hardware-oriented communication technologies[5]. Gubbi et al have outlined the capabilities of typical IoT devices as sensors and actuators, where sensors are triggered by changes in environment and take on a reporting capacity, and actuators receive commands and carry out motor or other similar tasks based on those commands[5]. Furthermore, their studies indicate that identification of individual devices is performed through assigned unique IDs or addresses on the physical network to which they are connected. Topology explanations by Friess et al in their book 'Vision for the Internet of Things' explains the various communication methodologies that can be implemented in way of bringing better cohesion between various IoT devices and the gateways through which they connect[6]. Some of these connectivity technologies are:

- Infra-red: Infrared is reliable and consumes a low amount of energy, but also has limited range
- Bluetooth: Standard Bluetooth connectivity, requires some set up and uses a large amount of energy but offers a large range
- Bluetooth Low-Energy (BLE): Uses less energy while providing the same amount of range as conventional Bluetooth[6]

Hong et al, in their development of an IP-based approach to IoT device connectivity, propose an alternate solution where the sensors and actuators themselves could connect directly over IP to the Internet, but also emphasize that this would increase the complexity of individual devices as opposed to having a simple device which connects via a gateway[7]. The general consensus among developers of IoT solutions seems to be that the best way to implement device-to-gateway communication is to design devices which are simple sensors or actuators, with connectivity to the Internet by interfacing with a gateway device, according to the generalization of IoT infrastructure and design methodologies outlined by Yang in her analysis of the Internet of Things and its various technologies[1]. The fact that gateways are the best solution is emphasized in the study done by Liu et al. as well[8], bringing up the importance of maintaining simple end devices in any successful IoT infrastructure. This further brings us to the problem of not being able to consider and control individual sensor devices connecting through multiple gateways.

2.2 IoT gateway architecture

Upon establishing that gateways to which devices can connect is the best solution available to build an IoT architecture, it is important to understand the underlying mechanisms that can be used in gateways to allow building a successful IoT solution. Gubbi et al., in their analysis of gateway devices, outline edge-processing enabled gateways which can manage and provide connectivity for devices while doing some operations required in terms of data processing and enrichment on the raw data received from sensor IoT devices[5]. According to Zhu et al., the gateway device is a bridge providing individual sensors and actuators with the connectivity and processing power that they cannot have on their own, empowering these devices and aggregating them to provide a single cohesive system[9]. It further provides a funnel through which streams of data from multiple sensor devices can be amalgamated to further solve many connectivity problems by using up less bandwidth than multiple devices transmitting individually. Freiss and his colleagues explain that the gateway device typically contains the following[6]:

- Protocol abstraction layer: Provides an interface for the gateway to communicate via standard IoT protocols such as MQTT and Zigby.
- Communication interface: A GSM or Wifi module, or both, which allows the gateway device to communicate with the Internet
- Control and power unit: Power unit which distributes power among the various components of the gateway device
- IoT Resource Management unit: Provides services such as device discovery and pairing mechanisms

IoT gateways typically connect to the Internet and send messages over one of a set of IoT messaging protocols. These protocols are designed to be lightweight in transport and also mostly based on the publisher/subscriber mode of design. Nakajima et al. state that it is important to select a lightweight protocol for gateway communication that also works over IP [10]. Figure 2.1 shows Yang's take on how different components of a typical IoT gateway interacts with the standard IP 7-layer architectural model[1]. In some cases, according to Friess et al., some components of the application layer may be present within a gateway device as well[6].

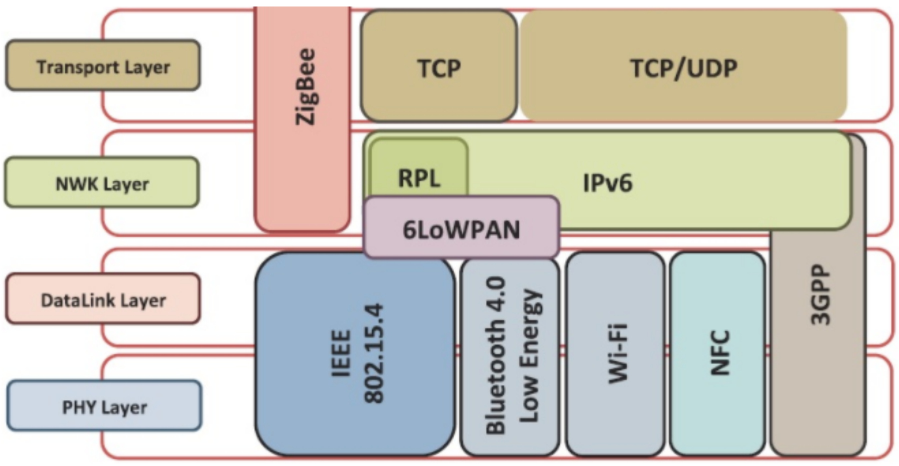


Figure 2.1: Gateway interaction with the layers of the Internet Protocol stack [1]

Typically, a cloud-based or server-hosted IoT broker server is required to receive messages from gateways and to enable management of those devices, from authentication and authorization to event and command handling.

2.3 IoT device management

In any IoT solution, the connecting devices, be they sensors/actuators or gateway devices, need to be tracked and managed. This is undertaken by the communications broker service that is the central messaging manager in any IoT architecture. According to Gubbi et al., the communications broker application can be hosted on a server used by the solution developer or be a Platform-as-a-Service (PaaS) application provided by a cloud vendor[5]. Yang provides the following list of tasks that must be carried out by a communications broker:

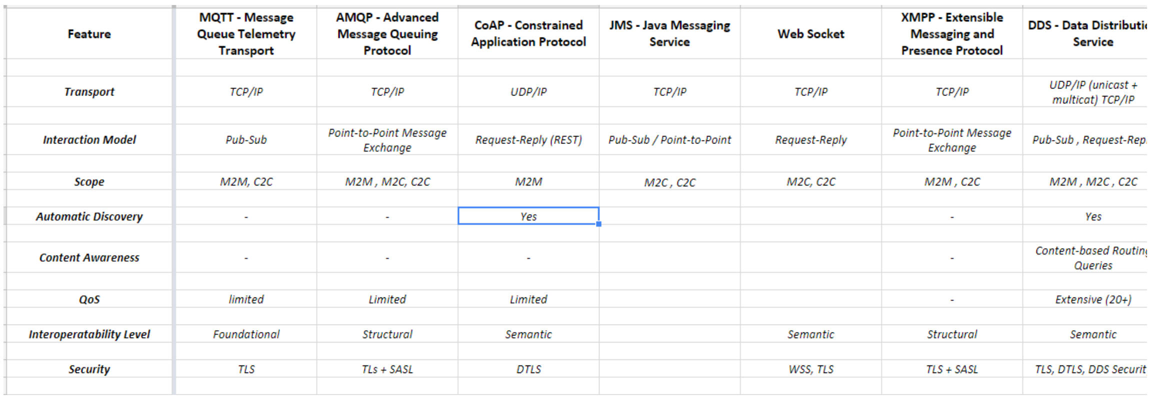
- Authorize and authenticate devices
- Receive events (data) from devices
- Send commands to devices

In addition to these, Hong et al. specify that a communications broker should be able to manage hardware-based functionality of devices such as firmware updates and remote reboots of devices etc[7].

The device management aspects of a communication broker's functionality focuses on the ability to register new devices on the communications broker so that they can connect, as well as managing the security aspects of each device[11]. According to Friess et al., IoT device management does not take in to account that individual sensor and actuator devices are connecting to the communications broker through a gateway, and considers the gateway itself as the only connecting device[6], making it complicated to manage individual sensor and actuator devices connecting through multiple gateways.

2.4 Comparison between IoT data protocols

Sarawi et al. outline the differences between the most popular IoT data protocols in the areas of transport mechanism, Quality of Service availability, interoperability levels and security [2]. They also discuss the importance of data protocols implementing features such as fault tolerance, content awareness and automatic discovery. A feature matrix based on this categorization can be found at 2.2.



Feature	MQTT - Message Queue Telemetry Transport	AMQP - Advanced Message Queuing Protocol	CoAP - Constrained Application Protocol	JMS - Java Messaging Service	Web Socket	XMPP - Extensible Messaging and Presence Protocol	DDS - Data Distributi Service
Transport	*TCP/IP*	*TCP/IP*	*UDP/IP*	*TCP/IP*	*TCP/IP*	*TCP/IP*	*UDP/IP (unicast + multicat) TCP/IP*
Interaction Model	*Pub-Sub*	*Point-to-Point Message Exchange*	*Request-Reply (REST)*	*Pub-Sub / Point-to-Point*	*Request-Reply*	*Point-to-Point Message Exchange*	*Pub-Sub , Request-Rep*
Scope	*M2M, C2C*	*M2M , M2C, C2C*	*M2M*	*M2C , C2C*	*M2C, C2C*	*M2M , C2C*	*M2M , M2C , C2C*
Automatic Discovery	-	-	*Yes*			-	*Yes*
Content Awareness	-	-	-			-	*Content-based Routin Queries*
QoS	*limited*	*Limited*	*Limited*			-	*Extensive (20+)*
Interoperatability Level	*Foundational*	*Structural*	*Semantic*		*Semantic*	*Structural*	*Semantic*
Security	*TLS*	*TLs + SASL*	*DTLS*		*WSS, TLS*	*TLS + SASL*	*TLS, DTLS, DDS Securit*

Figure 2.2: Feature matrix for copmarison of IoT messaging data protocols [2]

According to the feature matrix outlined at 2.2, MQTT shows promise as a robust and popular solution for the development of a typical IoT-based solution prototype.

2.5 Security issues faced in IoT networks and their implications

Security in IoT devices is a very important requirement of any IoT infrastructure [3]. Wan et al. define the security requirements of an IoT architecture to be in 4 layers:

- **Perceptual Layer**: This layer consists of the basic nodes of the IoT system, namely, the devices that are connecting to an IoT gateway. Since the nodes are low in power and storage capacity, it is difficult to employ security measures such as encryption. A collection of perceptual layer nodes could be vulnerable to be taken over and used in a denial-of-service attack. Lightweight cryptographic algorithms are needed on this layer to encrypt the transmission between nodes on this layer and the network, as well as maintaining a form of node authentication.

- **Network Layer**: The communication layer of the IoT architecture is vulnerable to man-in-the-middle attacks [3, 12]. This can be prevented through encryption of communication using standard protocols such as SSL and TLS for device-to-device or device-to-broker communications.

- **Support Layer**: The support layer consists of any intelligent processing resting on the cloud that is used to process and make sense of data sent by IoT devics. This layer needs to be secured using standard cloud security measures such as on-disk encryption and authenticated access [3].

- **Application Layer**: Applications that display processed data related to IoT platforms are vulnerable to various types of attacks that plague front-end systems [12]. These could also be used to steal device authentication data and to impersonate nodes on the system. Therefore the application layer needs to be secured using authentication and key agreement as well as education on security management to the human factor of the system.

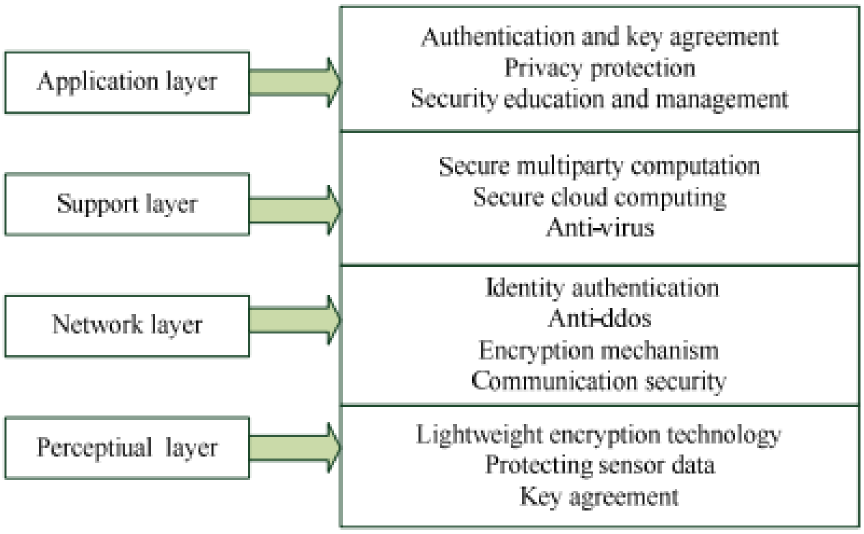


Figure 2.3: Security requirements in each layer of the IoT stack [3]

Figure 2.3 outlines the security requirements explained by Wan et al. for each layer they have identified on a typical IoT architecture.

Razzaq et al. outline the importance of maintaining security in IoT applications due to the abundance and numerous nature of IoT devices. In a distributed denial-of-service attack, millions of insecure IoT devices around the world could be used as zombies or impersonated effectively to bring down target websites. This was demonstrated in 2016 when a large number of compromised IoT devices were used to bring down a popular web hosting and domain name server provider known as Dyn [12]. This shows how important and essential security is in the forms of resilience to attacks, data authentication, access control and client privacy when considering an IoT application architecture.

2.6 Utilizing Blockchain as a secure, immutable storage mechanism

Blockchain is the online, decentralized, public ledger maintained as an immutable record of transactions, the most popular being that of Bitcoin[13]. As it is decentralized, there are many copies of any given Blockchain which can be maintained at different locations on peers from around the world. Each transaction is added to the ledger, which is then signed with a crypto key which can be discovered by solving a cryptographic problem. Maintainers of the Blockchain around the world will then attempt to decrypt the cryptographic problem in order to verify the authenticity of the transaction. Once it is verified by multiple parties, the transaction will be considered legitimate and passed on to the global Blockchain maintained across the world in its decentralized manner. This project explores the possibility of storing IoT device details on a block chain, enabling secure storage of individual device credentials without compromising the security of the whole IoT solution. Kosba et al., in their solution Hawk, which is utilizing the block chain to store smart contract details, emphasize on how the Blockchain can be used to secure data without having to implement new methods of cryptography while providing cryptographic primitives such as zero-knowledge proofs [14]. The current methodologies provided on Blockchain for securely storing data are as follows:

- Storing data as transactions on Blockchain itself
- Peer-to-peer file system e.g. InterPlanetary File System IPFS
- Decentralized cloud file storage e.g. Etherium Swarm
- Distributed databases e.g. Apache Cassandra

Out of these methodologies, most solutions which utilize Blockchain for secure storage will store data on the Blockchain itself, as it provides write once read only functionality that gives a globally-verifiable secure data storage mechanism[14][15]. Zyskind et al., in their proposal for using Blockchain as a platform for securely storing private data, also propose the use of Blockchain as direct storage for personal data such as passwords and payment method details, which validates the idea of storing authentication details for IoT devices on the Blockchain.

2.7 Hyperledger Blockchain

Hyperledger Fabric is identified as a viable candidate for the purposes of this project. Hyperledger Fabric is a Blockchain implementation that comprises of an immutable, peer-verified system of blocks which are cryptographically protected through multiple hashing mechanisms [4]. Hyperledger is a private and permissioned blockchain and not open, which also provides the facility to store information by creating new transactions and blocks through a consensus mechanism, rather than through proof of work, which would require a mining mechanism similar to what is seen on Bitcoin and Ethereum blockchains.

The Hyperledger Architecture Working Group states that the basic architecture of Hyperledger Fabric consists of the following components:

- Consensus Layer: Provides agreement mechanism between peers on the validity of transactions that make up a block on the blockchain
- Smart Contract Layer: Responsible for executing business logic to determine whether individual transactions are valid
- Communication Layer: Responsible for peer-to-peer transport of messages needed to be processed on a single blockchain instance
- Data Store Abstraction: Allows modularity in using different data storage mechanisms
- Crypto Abstraction: Modular layer that allows implementation of different methdologies for cryptography
- Identity Services: Provides root authorization when setting up a blockchain instance, also provides the mechanism to create new trust relationships within the blockchain
- Policy Services: Manages policies implemented on the system, such as consensus policy
- APIs: Allows other applications to interface with the blockchain
- Interoperation: Allows interaction and messaging between different blockchain instances

Hyperledger Fabric follows the concept of Consensus to validate transactions, which are handled through the consensus policies defined on the blockchain instance [4]. Consensus is achieved using Practical Byzantine Fault Tolerance (PBFT) which allows peers to 'vote' for the validity of a transaction. PBFT delivers transactions to peers at varying times, so that peers do not have the same order of transactions. Validating the order is also done through voting for a validating leader within the peer network. This ensures that rogue peer would be overruled by other other peers and disabled from disrupting or corrupting a transaction [4].

2.7.1 Transactions

According to the architecture outlined by Cachin, Hyperledger Fabric users will consist of validating peers and non-validating peers [16]. A validating peer is a node on the network that is responsible for maintaining and running consensus, validating transactions and maintaining the blockchain ledger. A non-validaing peer simply acts as a proxy from the outside world to interface with validating peers.

Each transaction that is added to a Hyperledger Fabric blockchain needs to gain consensus. Consensus is validated through smart contracts which are known as chaincode [16] and implemented on the Smart Contract Layer. Any transactions that cannot be validated through chaincode to reach consensus will be omitted from the blockchain deemed invalid by Fabric.

Once consensus is reached for a transaction through validation via smart contract, the individual transaction changeset will be communicated to the blockchain and added to a corresponding block. This process is outlined in Figure 2.4.

2.7.2 Immutability constraint and asset deletion

A typical IoT device management platform requires that devices which are added to the system should be deleted. Blockchain, by design, dictates that the transactions and blocks added to the blockchain cannot be modified or deleted. This poses an issue when managing assets such as devices that need to be modified or deleted on the blockchain. To solve this issue, Hyperledger Fabric provides an asset implementation through its AssetRegistry mechanism [17], which enables adding modifications and deletions to existing assets on the blockchain as new transactions, without modifying existing transactions. These new transactions point to asset IDs stored in asset creation transactions, allowing asset state to be maintained correctly.

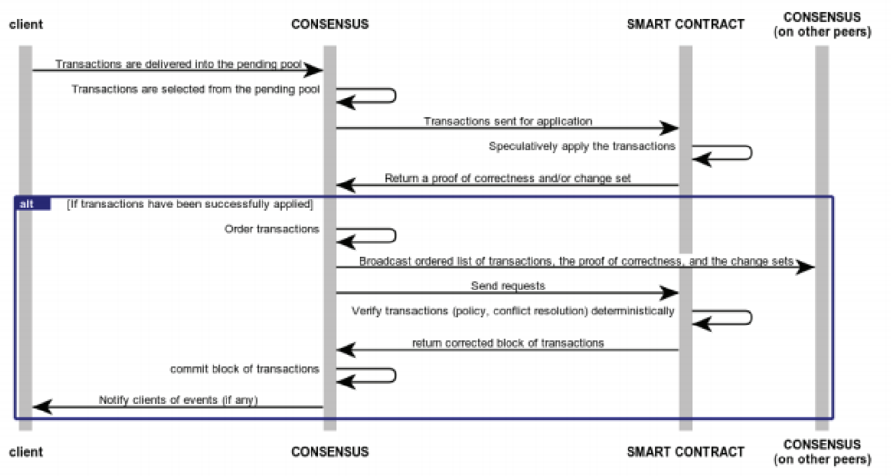

client
CONSENSUS
SMART CONTRACT
CONSENSUS (on other peers)
Transactions are delivered into the pending pool
Transactions are selected from the pending pool
Transactions sent for application
Speculatively apply the transactions
Return a proof of correctness and/or change set
alt
[If transactions have been successfully applied]
Order transactions
Broadcast ordered list of transactions, the proof of correctness, and the change sets
Send requests
Verify transactions (policy, conflict resolution) deterministically
return corrected block of transactions
commit block of transactions
Notify clients of events (if any)
client
CONSENSUS
SMART CONTRACT
CONSENSUS (on other peers)

Figure 2.4: State diagram depicting Consensus in a Hyperledger Blockchain [4]

2.8 Summary

The following details were finalized as the result of this literature review:

- A typical IoT infrastructure will make use of sensor and actuator devices which will connect to the Internet through a gateway device
- While it is possible to connect a sensor or actuator device directly to the Internet, for complexity reasons, it is recommended to connect via gateway device
- Multiple individual IoT devices connecting to the Internet will not be identified as such as they will be identified by the gateway device that they are connecting through, making them hard to manage
- A device management framework that handles device creation, updates and removal as well as processing messages sent by devices and the sending of commands to the devices over a standard IoT protocol while maintaining credential security would provide the required solution for remote IoT device management
- MQTT is recognized as a viable (and most frequently used) communication protocol for a modern IoT solution
- Security is a major component within any IoT application, as compromised devices or nodes could lead to major security issues
- The Blockchain can be utilized as secure storage for private data such as IoT device details and their authentication credentials

Chapter 3

Analysis and design

This chapter describes the design of the IoT broker client, alongside the implementation of Blockchain as a secure storage catalog provider to any implementation of an IoT solution which utilizes a basic MQTT broker platform.

3.1 Top-level architecture of the system

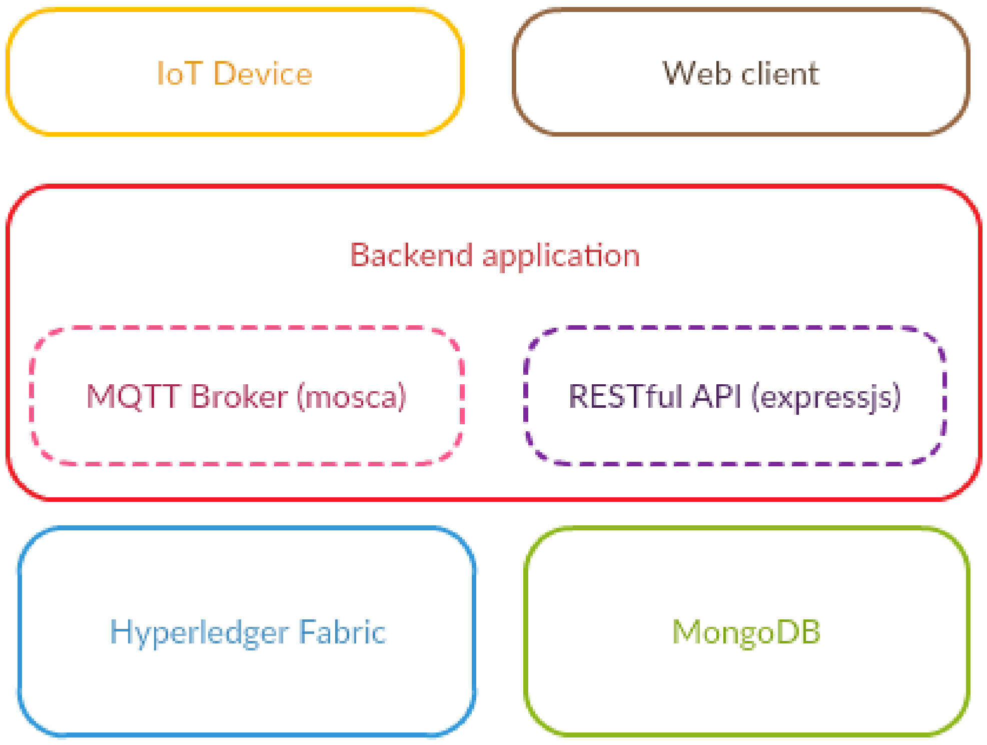


Figure 3.1: Block diagram outlining the solution architecture

The top-level architecture of the system is outlined in Figure 3.1. The architecture of the system consists of the following components:

- Backend application: The backend application consists of an IoT broker which acts as a messaging server for MQTT messages and device connectivity, and a RESTful API application which caters to the web client for device management and message viewing.

- Hyperledger Fabric blockchain: The blockchain implementation is used as a secure storage layer for device details including credentials and to store messages sent by IoT devices. MongoDB is used as the associated database.
- Web client: A web client is used to provide an interface for users to interact with the system, to create devices and to view messages sent by devices.

3.2 Design of the backend application (IoT Broker + RESTful API)

The backend application consists of an IoT broker server application and a RESTful API.

In order to emulate a typical IoT broker scenario, an MQTT broker application was developed. MQTT was identified as the most frequently used messaging protocol in modern IoT [1] and as such, was selected as the messaging protocol to be used for the devices and the broker platform to communicate, providing a generic platform for integration with the blockchain backend.

The generic message broker developed provides the following features:

- MQTT 3.1 and 3.1.1 compliance
- Support for Quality of Service levels 0 and 1 outlined in the MQTT specification
- Support for Transport Layer Security version 1.2
- Integration with Hyperledger Fabric for credential, device detail and message storage
- Acts as a backend RESTful API server for the device management frontend web application

In addition to brokering MQTT messages, the backend application provides RESTful API support for the following tasks:

- Device creation, updates and deletes
- Creating device owners (users) on the system
- Managing devices under a single device owner
- Viewing messages that have been sent from devices

The endpoints designed for the device management API, user management API and the messaging API can be seen in Figures 3.2, 3.3 and 3.4 respectively.

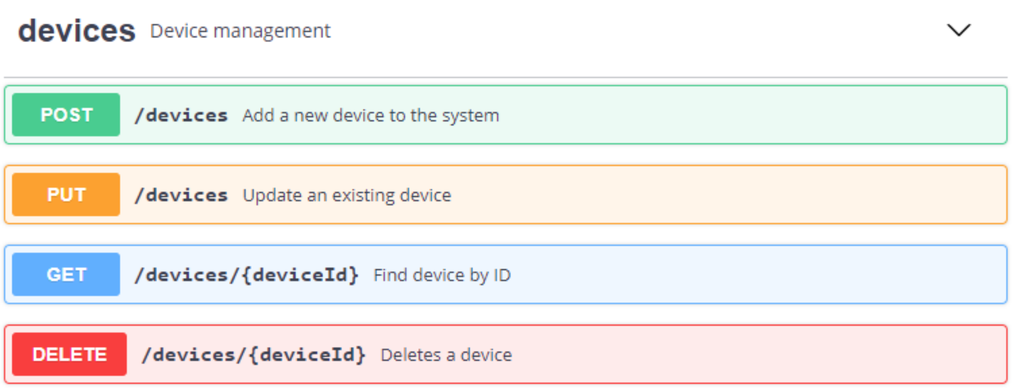
The IoT MQTT broker also provides a chaincode client integration to create transactions on the Hyperledger Fabric blockchain through a custom implementation of the Fabric NodeJS SDK.

3.3 Hyperledger Fabric as secure storage

Fabric provides the creation of transactions on the blockchain through a client implementation of chaincode, the smart contract system of Fabric. The chaincode for this project is implemented similar to a typical asset management system as outlined by Hyperledger documentation.

The device management system on Fabric, at the top level, contains the following entities:

- Device



devices Device management

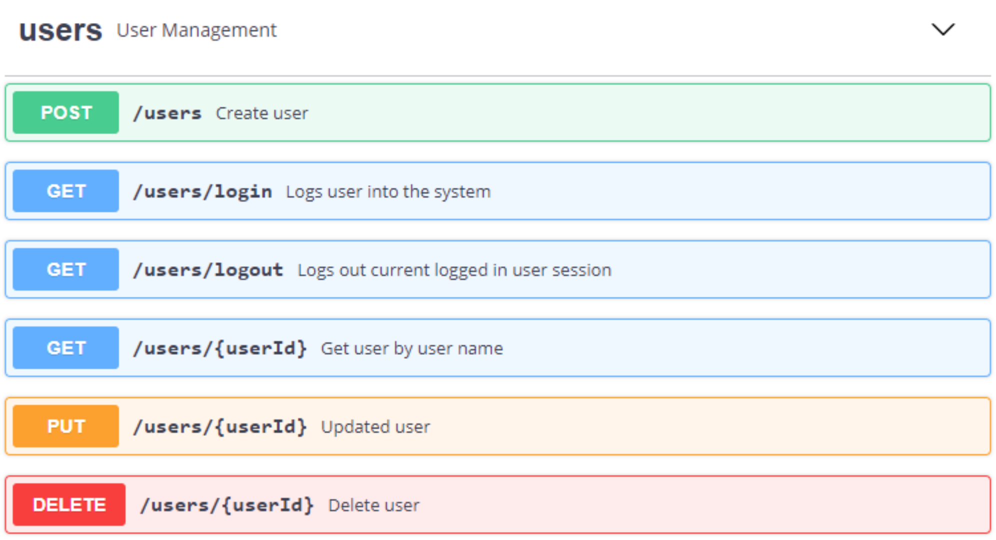
POST /devices Add a new device to the system

PUT /devices Update an existing device

GET /devices/{deviceId} Find device by ID

DELETE /devices/{deviceId} Deletes a device

Figure 3.2: API endpoints for device management



users User Management

POST /users Create user

GET /users/login Logs user into the system

GET /users/logout Logs out current logged in user session

GET /users/{userId} Get user by user name

PUT /users/{userId} Updated user

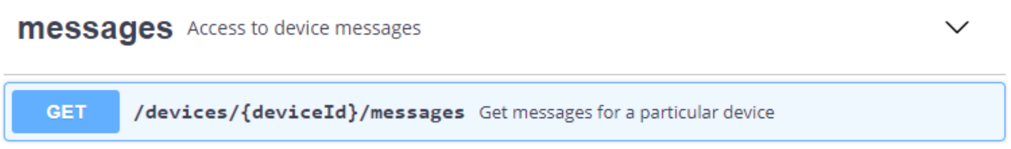
DELETE /users/{userId} Delete user

Figure 3.3: API endpoints for user management

- DeviceOwner
- DeviceMessage

The following methods are implemented on chaincode to maintain transactions on the blockchain to correspond to actions performed using the system:

- createDevice: Creates a transaction corresponding to a new device on Fabric along with its credentials
- modifyDevice: Uses AssetRegistry functionality on the system to modify an existing Device asset entity
- deleteDevice: Uses AssetRegistry to remove an existing device asset entity
- createDeviceMessage: Creates a transaction on Fabric corresponding to a message sent to the system over MQTT from a device, along with sensor readings/events sent by the device
- createDeviceCommand: Creates a transaction on Fabric corresponding to a command that has been sent to a device over MQTT by the system


messages Access to device messages

GET /devices/{deviceId}/messages Get messages for a particular device

Figure 3.4: API endpoints for messaging

Consensus on the Fabric implementation for the system is achieved through verification on multiple peers view PBFT, and as such, the system requires at least 2 peer nodes to function. Fabric provides a modular architecture with the ability to change the storage module. This project utilizes the MongoDB storage module and Fabric can therefore use a MongoDB database to store data.

3.4 Web client

The web client is developed with the aim of providing a user interface to interact with the RESTful APIs for the tasks for user authentication, user management, device management and message viewing. The following screens are included in the web UI:

- Login screen: Interacts with the login API of the backend
- Device management screen: Provides UI to view list of devices, add/update devices and to delete devices
- Device message list screen: Provides UI to view messages sent by a particular device

3.5 Process flow

The process of registering devices on the system via chaincode invocation is outlined in Figure 3.5.

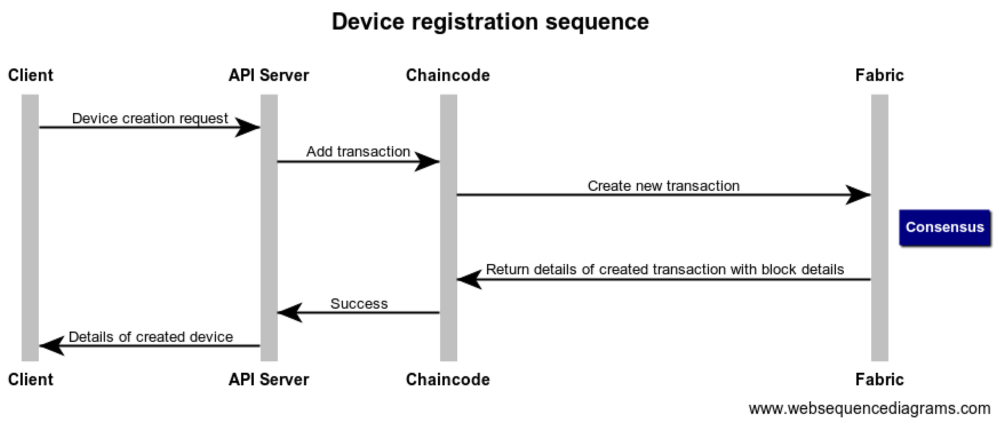


Figure 3.5: Device registration sequence diagram

The process of authenticating devices via the credentials stored on blockchain and subsequently receiving and storing messages on Fabric via chaincode invocation is outlined in Figure 3.6.

The sequence diagram shows how there are two round trips through Fabric to perform authentication, where the backend application will send out user credentials for authentication on

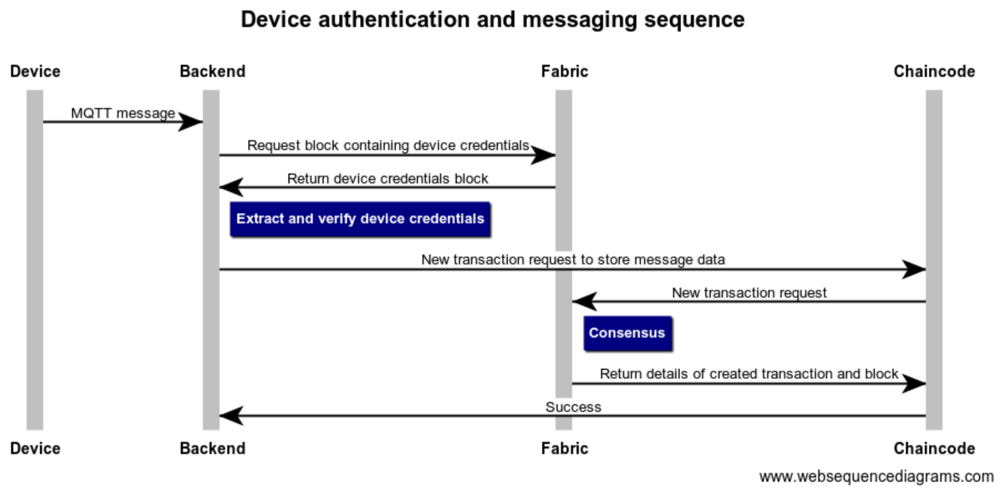

Device authentication and messaging sequence
Device
Backend
Fabric
Chaincode
MQTT message
Request block containing device credentials
Return device credentials block
Extract and verify device credentials
New transaction request to store message data
New transaction request
Consensus
Return details of created transaction and block
Success
Device
Backend
Fabric
Chaincode
www.websequencediagrams.com

Figure 3.6: Device authentication and message receiving sequence diagram

the private blockchain. User creation is also done through the user management mechanisms and identity services provided by Fabric, which enables effective authentication and authorization. As messages are stored as new transactions on blocks, complete non-repudiation of information is ensured due to the immutable nature of the blockchain.

Chapter 4

Implementation

This chapter outlines the implementation details of the various components outlined in the previous chapter.

4.1 Blockchain

The blockchain implementation used is Hyperledger Fabric, using MongoDB for persistence. The initial implementation follows a 4-peer node architecture, with the following components:

- 4 peer nodes: Fabric nodes acting as peers for the blockchain implementation, maintaining copies of the blockchain and providing consensus for new transactions.

- Certificate authority: The certificate authority acts as the point for generating trust relationships between peers and the blockchain. It also generates trust certificates for administrators.

- Ordering service: The ordering service acts as the communication method for the blockchain and peers. It provides channels for different peers to communicate on. The nature of this communication ranges from reading the current state of the ledger to creating new transactions

The blockchain implementation also contains a chaincode (smart contract) application which enables interfacing with the ledger over a RESTful interface. This allows the NodeJS SDK to create new transactions and extract transaction details from the blockchain and also supports the asset tracking mechanism which allows devices to be stored as assets on Hyperledger Fabric. All applications on the blockchain (individual peers and supporting applications) are deployed within Docker containers for better management of deployment environments and containerization.

4.2 Backend application - XylemJS

The backend application (named XylemJS) consists of an IoT message broker as well as a RESTful web service application. The application also provides the integration with the Fabric blockchain via the NodeJS SDK and the chaincode implementation on Golang. The following sections outline the technologies used in the implementation of the backend application, along with the motivations behind selecting each technology.

4.2.1 NodeJS

NodeJS provides a non-blocking implementation of a server which allows for fast parallel processing out of the box. This allows the creation of applications that can asynchronously handle multiple input output queues without blocking the execution thread. In the scenario explored in this research, the backend application is required to be able to connect to multiple IoT devices simultaneously, validate them and receive messages from them while maintaining a connection with the Hyperledger Fabric blockchain. Therefore, NodeJS was selected as the programming language for the backend application for its capability to handle concurrency.

4.2.2 ExpressJS

ExpressJS is a Model-View-Controller framework designed to work with NodeJS. It provides the capability of hosting a server with an externally-open application programming interface. It also provides the capability for interactions with other applications and the impelemntation of RESTful web services over HTTP. ExpressJS was selected to implement the RESTful web services on the backend application due to its capability of integrating seamlessly with other NodeJS libraries and to provide RESTful web service endpoints.

4.2.3 Mosca

Mosca is a IoT broker library for NodeJS that provides an event-driven interface for handling MQTT messages over predefined topics. It also provides a level of modularity in terms of defining a storage mechanism. Mosca was selected to implement the IoT broker due to its integration with NodeJS and the capability of developing a module for Mosca that uses blockchain as the storage mechanism for device messages sent on MQTT topics.

4.2.4 Topics and commands

This section outlines the definition of the MQTT messages that are used for communication with the XylemJS platform. MQTT messages come with subscription topic implementations which are used as generic topics defining the message types.
The MQTT messages sent to and from the platform can be categorized as follows:

Table 4.1: Supported MQTT message types

Message type	Description	Generic topic
Events	Events are data sent from devices related to values obtained through sensors and other methods on the device. Devices could also send health related information pertaining to the various components on each device. Data is received by XylemJS and stored accordingly on the Fabric blockchain component where they can be extracted and processed via the RESTful APIs	xylem/event/deviceId/event-type
Commands	Commands are issued by the XylemJS platform on user request to issue execution commands to devices connected to the platform, which may include activating actuators	xylem/deviceId/command-type

4.2.5 AngularJS

The web application provides a user interface for users to interact with the RESTful web services provided by the backend application. This enables the user to manage devices and view device messages that have been sent to the backend over MQTT, as well as to generate commands to be sent to the devices so that they can be controlled remotely.

AngularJS is a Model-View-View-Model framework for JavaScript that provides dynamic data binding and event handling through and event loop on the frontend. it is a popular framework used to develop fast, scalable single-page applications. AngularJS was chosen as the frontend technology for the system due to its capability to interface efficiently with RESTful web services provided by the backend application, and its maintainability due to its well-structured nature.

Chapter 5

Evaluation

5.1 Research hypotheses

As the project involves the creation of a blockchain-based IoT device management solution, the following hypotheses were formulated:

- Blockchain can provide a secure storage mechanism for IoT device credentialls
- It is possible to use blockchain as a tracking mechanism for device management
- It is possible to use blockchain as a medium of separation of assigned devices between users

The outlined evaluation approach tests the above hypotheses. As this project involves the development of an end-user solution, the approach undertaken for evaluation is of an experiment-based nature.

5.2 Evaluation approach

The evaluation approach an experiment-based nature, involving evaluation through automated testing taking user roles and user stories in to consideration. It is achieved through an automated Behavioral Tests using a BTDD (Behavioral Test Driven Development) framework. the hypotheses are proven through passing of these tests to an acceptable level.

Through the evaluation of the hypotheses by attaching them to specific user stories, the solution is tested through scenarios that would arise in a live deployment. This enables verification whether the solution is production-ready.

The user stories outlined for the behavioral test are as follows:

Table 5.1: User stories for evaluation

User (As a...)	Story
Administrator	I should be able to create new chaincode channels
Authorized user	I should be able to authenticate myself to view the blockchain
Authenticated user	I should be able to add credentials for new devices to the chaincode
Authenticated user	I should be able to view details of registered devices from the chaincode
Unauthorized user	I should not be able to access details regarding any devices managed through the system
Unauthorized user	I should not be able to access any transactions on the blockchain
Authenticated user	I should not be able to view details of devices not belonging to me

The automated behavioral tests, once written, will be analyzed for levels of coverage for the above user stories using a technique of elimination; taking in to account all possible scenarios involved in each user story.

The results of the evaluation will depend on the capability of the solution to pass the behavioral tests with a maximum success rate.

5.3 Outline of behavioral test mechanism

Behavior-Driven Development(BDD) focuses on recreating the user's requirements exactly in the form of tests designed and executed in languages that focus on emulating simple spoken language such as Gherkin and Cucumber.

The tests designed to evaluate this project adhere to the following general form:

```
Feature: Login

  Scenario: I want to login as a device owner
    Given I am on the "Login"
    When I fill "username" field with "username@example.com"
    And I fill "password" field with "pass"
    And I click on the button "Login"
    And I should be redirected on "Console"
```

A test coverage report that can be obtained through the BDD framework will be attached as part of the evaluation results, alongside the results of the tests.

5.4 Results

The results of the evaluation are as follows:

Table 5.2: Evaluation results

User (As a...)	Story	Test Result
Administrator	I should be able to create new chaincode channels	PASS
Authorized user	I should be able to authenticate myself to view the blockchain	PASS
Authenticated user	I should be able to add credentials for new devices to the chaincode	PASS
Authenticated user	I should be able to view details of registered devices from the chaincode	PASS
Unauthorized user	I should not be able to access details regarding any devices managed through the system	PASS
Unauthorized user	I should not be able to access any transactions on the blockchain	PASS
Authenticated user	I should not be able to view details of devices not belonging to me	PASS

Chapter 6

Conclusion

This chapter takes into account the solution discussed in this dissertation, its analysis and design, implementation and evaluation and discusses the viability of using the solution to the stated problem. It also provides an executive summary of the main points outlined in each chapter.

6.1 Introduction

The concepts related to IoT and blockchain are discussed, presenting the various architectural layers of the IoT stack, and the core features of distributed ledger technology and blockchain.

The goal of the project is described as follows: Integrate with a typical IoT broker framework to ensure that IoT devices (sensors and actuators) connecting to the gateway component of the framework can be controlled regardless of physical gateway they are connecting through, and provide secure storage for all device details and messages on a private blockchain.

The scope and the assumptions related to the implementation of the solution are discussed, the scope alludes to the design and development of an IoT broker framework which uses blockchain for storage.

6.2 Analysis and Design

This chapter takes in to account the main components required for the development of the solution, which are as follows:

- Backend application: The backend application consists of an IoT broker which acts as a messaging server for MQTT messages and device connectivity, and a RESTful API application which caters to the web client for device management and message viewing.
- Hyperledger Fabric blockchain: The blockchain implementation is used as a secure storage layer for device details including credentials and to store messages sent by IoT devices. MongoDB is used as the associated database.
- Web client: A web client is used to provide an interface for users to interact with the system, to create devices and to view messages sent by devices.

6.3 Implementation

The implementation details of the various components described in the earlier chapter are examined in this chapter. The technologies used and their viability is also discussed. The components discussed are:

- Blockchain storage: Implemented using Hyperledger Fabric
- Backend application (XylemJS): Implemented using NodeJS, ExpressJS and Mosca
- Frontend web application: Implemented using AngularJS

6.4 Evaluation

This chapter presents the research hypotheses and the evaluation methodology for validation of that hypotheses. The research hypotheses explored are:

- Blockchain can provide a secure storage mechanism for IoT device credentialls
- It is possible to use blockchain as a tracking mechanism for device management
- It is possible to use blockchain as a medium of separation of assigned devices between users

The tests carried out on the developed solution using Behavior-Driven Development and Acceptance Test-Driven Development methodologies are outlined along with their results in this chapter.

As per the evaluation, all research hypotheses are validated.

References

[1] S. H. Yang. The internet of things. *Wireless Sensor Networks*, 29(7):241–267, 2013.

[2] K. Aleiyan M. Alzubaidi S. Sarawi, M. Anbar. Internet of things communication protocols: Review. In *8th National Conference on Information Technology*, 2017.

[3] J. Wan H. Suo. Security in the internet of things: A review. In *2012 International Conference on Computer Science and Electronics Engineering*, 2012.

[4] Hyperledger Architecture Working Group. *Hyperledger Architecture, Volume 01*. Hyperledger Community, 2017.

[5] S. Marusic M. Palaniswami J. Gubbi, R. Buyya. The internet of things (iot): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, September 2013.

[6] P. Freiss S. Woellfle H. Sundmaekar, P. Guillemin. *Vision and Challenges for Realising the Internet of Things*. Publications office of the European Union, 2010.

[7] M. Ha S. Bae S. J. Park W. Jung J. Kim S. Hong, D. Kim. Snail: an ip-based wireless sensor network approach to the internet of things. *IEEE Wireless Communications*, 17(6), 2010.

[8] G. Zhou Y. Liu. Key technologies and applications of internet of things. In *2012 Fifth International Conference on Intelligent Computation Technology and Automation (ICICTA)*, 2012.

[9] Q. Chen Y. Liu W. Qin Q. Zhu, R. Wang. Iot gateway: Bridging wireless sensor networks into internet of things. In *IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing (EUC)*, 2010.

[10] E. Tokunaga F. Stajano T. Nakajima, H. Ishikawa. Technology challenges for building internet-scale ubiquitous computing. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems, 2002*, 2002.

[11] J. Jiang G. Gan, Z. Lu. Internet of things security analysis. In *2011 International Conference on Internet Technology and Applications (iTAP)*, 2011.

[12] S. Gill S. Ullah M. Razzaq, M. Qureshi. Security issues in the internet of things (iot): A comprehensive study. In *(IJACSA) International Journal of Advanced Computer Science and Applications*, 2017.

[13] S. Nakamoto. *Bitcoin: A Peer-to-Peer electronic cash system*. UNICAMP–IA368, 2013.

[14] E. Shi Z. Wen C. Papamanthou A. Kosba, A. Miller. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, 2016.

[15] A. Pentland G. Zyskind, O. Nathan. Decentralizing privacy: Using blockchain to protect personal data. In *2015 IEEE Security and Privacy Workshops (SPW)*, 2015.

[16] C. Cachin. Architecture of the hyperledger blockchain fabric. Technical report, IBM Research, Zurich, 2016.

[17] M. Vukolic J. Sousa, A. Bessani. *A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform.* 2017.

Appendix A

Important source code

A.1 XylemJS

A.1.1 DeviceController.js

```
let self;

export default class DeviceController {
    constructor(express, deviceService, deviceMessageService, constants, helpersUtil) {
        self = this;
        self.expressRouter = new express.Router({ mergeParams: true });
        self.deviceService = deviceService;
        self.deviceMessageService = deviceMessageService;
        self.constants = constants;
        self.helpersUtil = helpersUtil;

        self.expressRouter.post('', self.createDevice);

        self.expressRouter.put('/:id', self.updateDevice);

        self.expressRouter.get('/:id', self.findDevice);

        self.expressRouter.delete('/:id', self.removeDevice);

        self.expressRouter.get('/:id/messages', self.getDeviceMessages);

        return self.expressRouter;
    }

    createDevice(req, res, next) {
        let device = {
            clientId: req.body.clientId,
            type: req.body.type,
            ownerId: req.body.ownerId,
            token: req.body.token
        };
        self.deviceService.insertDevice(device)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    findDevice(req, res, next) {
        self.deviceService.findDevice(req.params.id)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }


    updateDevice(req, res, next) {
        self.deviceService.updateDevice(req.params.id, req.body)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    removeDevice(req, res, next) {
        self.deviceService.removeDevice(req.params.id)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    getDeviceMessages(req, res, next) {
        self.deviceMessageService.findMessagesByDevice(req.params.id)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

}
```

A.1.2 DeviceMessageController.js

```
let self;

export default class DeviceMessageController {
    constructor(express, deviceMessageService, constants, helpersUtil) {
        self = this;
        self.expressRouter = new express.Router({ mergeParams: true });
        self.deviceMessageService = deviceMessageService;
        self.constants = constants;
        self.helpersUtil = helpersUtil;

        self.expressRouter.get('', self.getDeviceMessages);

        return self.expressRouter;
    }

    getDeviceMessages(req, res, next) {
        self.deviceMessageService.findMessagesByDevice(req.params.id)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

}
```

A.1.3 DeviceService.js

```
let self;

/**
 * Device Service
 * @constructor
 */
export default class DeviceService {
    constructor(config,
                constants,
                exceptionFactory,
                q,
                deviceRepository) {

        self = this;
        self.q = q;
        self.config = config;
        self.constants = constants;
        self.exceptionFactory = exceptionFactory;
        self.deviceRepository = deviceRepository;
    }
    insertDevice(device) {
        return self.deviceRepository.insertDevice(device)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }

    findDevice(id) {
        return self.deviceRepository.findDevice(id)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }

    updateDevice(id, device) {
        return self.deviceRepository.updateDevice(id, device)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }

    removeDevice(id) {
        return self.deviceRepository.removeDevice(id)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }
}
```

A.1.4 DeviceMessageService.js

```
let self;

/**
 * Device Message Service
 * @constructor
 */
export default class DeviceMessageService {
    constructor(config,
                constants,
                exceptionFactory,
                q,
                deviceMessageRepository) {

        self = this;
        self.q = q;
        self.config = config;
        self.constants = constants;
        self.exceptionFactory = exceptionFactory;
        self.deviceMessageRepository = deviceMessageRepository;
    }

    insertDeviceMessage(deviceId, topic, payload) {
        let message = {
            topic: topic,
            payload: payload
        };
        return self.deviceMessageRepository.insertDeviceMessage(deviceId, message)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }

    findDeviceMessage(deviceId, messageId) {
        return self.deviceMessageRepository.findDeviceMessage(deviceId, messageId)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }

    findMessagesByDevice(deviceId) {
        return self.deviceMessageRepository.findMessagesByDevice(deviceId)
            .then((result) => {
                let messages = [];
                if(Array.isArray(result)) {
                    for(let i = 0; i < result.length; i++) {
                        let message = result[i];
                        if(message.topic.split("/")[0] !== "$SYS") {
                            messages.push(message);
                        }
                    }
                }
                return self.q.when(messages);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(null);
            });
    }
}
```

A.1.5 FabricDeviceRepository.js

```
'use strict';
import BaseRepository from './BaseRepository';

let self, schema;

export default class FabricDeviceRepository extends BaseRepository {
    constructor(q, config, constants, fabclient) {
        super(q, config, constants, config.dataModel.collection.device);
        self = this;
        self.q = q;
        self.config = config;
        self.constants = constants;
        self.fabclient = fabclient;


    }

    insertDevice(device) {
        return self.fabclient._insert(device, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }

    findDevice(id) {
        let query = {
            clientId: id
        };

        return self.fabclient._find(query, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }

    findDevicesByUser(userId) {
        let query = {
            ownerId: userId
        };
        return self.fabclient._findAll(query, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }

    updateDevice(id, updateDevice) {
        let query = {
            clientId: id
        };
        return self.fabclient._update(query, updateDevice, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }

    removeDevice(id) {
        let deleteDevice = {
            clientId: id
        };

        return self.fabclient._remove(deleteDevice, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }
}
```

A.1.6 FabricDeviceMessageRepository.js

```
'use strict';
import BaseRepository from './BaseRepository';

let self, schema;

export default class DeviceMessageRepository extends BaseRepository {
    constructor(q, config, constants, fabclient) {
        super(q, config, constants, config.dataModel.collection.deviceMessage);
        self = this;
        self.q = q;
        self.config = config;
        self.constants = constants;
        self.fabclient = fabclient;

    }

    insertDeviceMessage(deviceId, message) {
        message.deviceId = deviceId;
        return self.fabclient._insert(message, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }

    findDeviceMessage(deviceId, messageId) {
        let query = {
            _id: messageId,
            deviceId: deviceId
        };

        return self.fabclient._find(query, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }

    findMessagesByDevice(deviceId) {
        let query = {
            deviceId: deviceId
        };
        return self.fabclient._findAll(query, schema)
            .then((result) => {
                return result;
            })
            .catch((err) => {
                console.log(err);
                return self.q.when({});
            });
    }
}
```

A.1.7 UserController.js

```
let self;

export default class UserController {
    constructor(express, userService, constants, helpersUtil) {
        self = this;
        self.expressRouter = new express.Router();
        self.userService = userService;
        self.constants = constants;
        self.helpersUtil = helpersUtil;

        self.expressRouter.post('', self.createUser);

        self.expressRouter.get('', self.getUsers);

        self.expressRouter.put('/:id', self.updateUser);

        self.expressRouter.get('/:id', self.findUser);

        self.expressRouter.delete('/:id', self.removeUser);

        self.expressRouter.get('/:id/devices', self.getDevicesByUser);

        return self.expressRouter;
    }

    createUser(req, res, next) {
        let user = {
          firstName: req.body.firstName,
          lastName: req.body.lastName,
          userName: req.body.userName,
          password: req.body.password,
          role: req.body.role
        };
        self.userService.insertUser(user)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }
    getUsers(req, res, next) {
        self.userService.getUsers()
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    findUser(req, res, next) {
        self.userService.findUser(req.params.id)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    updateUser(req, res, next) {
        self.userService.updateUser(req.params.id, req.body)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    removeUser(req, res, next) {
        self.userService.removeUser(req.params.id)
            .then((result) => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }

    getDevicesByUser(req, res, next) {
        self.userService.getDevicesByUser(req.params.id)
            .then(result => {
                res.status(self.constants.SUCCESS).json(result);
            })
            .catch((err) => {
                return next(err);
            });
    }
}
```

A.1.8 UserService.js

```
let self;

/**
 * User Service
 * @constructor
 */
export default class UserService {
    constructor(config,
                constants,
                exceptionFactory,
                q,
                userRepository,
                deviceRepository) {

        self = this;
        self.q = q;
        self.config = config;
        self.constants = constants;
        self.exceptionFactory = exceptionFactory;
        self.userRepository = userRepository;
        self.deviceRepository = deviceRepository;
    }

    insertUser(user) {
        return self.userRepository.insertUser(user)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(err);
            });
    }

    findUser(id) {
        return self.userRepository.findUser(id)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(err);
            });
    }

    getUsers() {
        return self.userRepository.getUsers()
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(err);
            });
    }

    updateUser(id, user) {
        return self.userRepository.updateUser(id, user)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(err);
            });
    }

    removeUser(id) {
        return self.userRepository.removeUser(id)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(err);
            });
    }

    insertBulkUsers(usersArray) {
        return self.userRepository.insertBulkUsers(usersArray)
            .then((result) => {
                return self.q.when(result);
            })
            .catch((err) => {
                console.log(err);
                return self.q.when(err);
            });
    }

    getDevicesByUser(userId) {
        return self.deviceRepository.findDevicesByUser(userId)
            .then(result => {
                return self.q.when(result);
            })
            .catch(err => {
                console.log(err);
                return self.q.when(err);
            });
    }

    loginUser(username, password) {
        return self.userRepository.getUserByUsername(username)
            .then(result => {
                if(!result || !result.username) {
                    throw self.exceptionFactory.createInstance('E0100', 404);
                }
                return self.userRepository.checkPassword(username, password)
                    .then(user => {
                       if(user && user.username) {
                           delete user.password;
                           return self.q.when(user);
                       } else {
                           throw self.exceptionFactory.createInstance('E0101', 403);
                       }
                    });
            })
            .catch(err => {
                console.log(err);
                return self.q.when(err);
            });
    }
}
```

A.1.9 UserRepository.js

```
'use strict';
import BaseRepository from './BaseRepository';

let self, schema;

export default class UserRepository extends BaseRepository {
    constructor(q, config, constants, mongoose) {
        super(q, config, constants, config.dataModel.collection.user);
        self = this;
        self.q = q;
        self.config = config;
        self.constants = constants;
        self.mongoose = mongoose;

        let Schema = self.mongoose.Schema;
        let schemaStructure = {
            username: String,
            password: String,
            firstName: String,
            lastName: String,
            role: String
        };
        schema = self.mongoose.Schema(schemaStructure, { collection: self.config.dataModel.c
    }

    insertUser(user) {
        return self._insert(user, schema)
            .then((result) => {
```

```
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

getUsers() {
    return self._findAll({}, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

findUser(id) {
    let query = {
        _id: id
    };

    return self._find(query, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

updateUser(id, updateUser) {
    let query = {
        _id: id
    };
    return self._update(query, updateUser, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

removeUser(id) {
    let deleteUser = {
        _id: id
    };

    return self._remove(deleteUser, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

insertBulkUsers(users) {

    return self._bulkInsert(users, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

getUserByUsername(username) {
    let query = {
        username: username
    };

    return self._find(query, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
            return self.q.when({});
        });
}

checkPassword(username, password) {
    let query = {
        username: username,
        password: password
    };

    return self._find(query, schema)
        .then((result) => {
            return result;
        })
        .catch((err) => {
            console.log(err);
```

```
                return self.q.when({});
            });
    }
}
```

A.1.10 BaseRepository.js

```
'use strict';
/**
 * Base class for repository pattern
 */

export default class BaseRepository {
    constructor(q, config, constants, collection) {
        this.q = q;
        this.config = config;
        this.constants = constants;
        this.collection = collection;
    }

    _insert(object, schema) {
        this.model = this.mongoose.model(this.collection, schema);

        let model = new this.model(object);
        let insert = this.q.nbind(model.save, model);

        return insert()
            .then((result) => {
                return result;
            });
    }

    _find(query, schema) {
        this.model = this.mongoose.model(this.collection, schema);

        let find = this.q.nbind(this.model.findOne, this.model);

        return find(query)
            .then((result) => {
                return this.q.when(result);
            });
    }

    _findAll(query, schema) {
        this.model = this.mongoose.model(this.collection, schema);

        let find = this.q.nbind(this.model.find, this.model);
        return find(query)
            .then((result) => {
                return this.q.when(result);
            });
    }
    _remove(query, schema) {
        this.model = this.mongoose.model(this.collection, schema);

        let remove = this.q.nbind(this.model.remove, this.model);

        return remove(query)
            .then((result) => {
                return this.q.when(result);
            });
    }

    _update(query, updateDoc, schema) {
        this.model = this.mongoose.model(this.collection, schema);

        let update = this.q.nbind(this.model.update, this.model);

        return update(query, updateDoc)
            .then((result) => {
                return this.q.when(result);
            });
    }

    _bulkInsert(docs, schema) {
        this.model = this.mongoose.model(this.collection, schema);

        let insert = this.q.nbind(this.model.insertMany, this.model);

        return insert(docs)
            .then((result) => {
                return this.q.when(result);
            });
    }
}
```

A.1.11 mqttsv.js

```
import mosca from 'mosca';
import config from './config/Configuration';
import container from './ConfigIoc';

let server;
let deviceMessageService = container.resolve('deviceMessageService');
let ascoltatore = {
    //using ascoltatore
    type: 'mongo',
    url: config.DB.mongodb.nodebaseapp.connection,
    pubsubCollection: config.dataModel.collection.pubSub,
    mongo: {}
};

let moscaSettings = {
    // port: 1883,
    interfaces:[
        {
            type:"mqtt",
            port: 1883
        },
        // {type:"mqtts", port:8883, credentials:{keyPath:SECURE_KEY,certPath: SECURE_CERT}
    ],
    backend: ascoltatore,
    logger:{
        name: "secureExample",
        label: 40,
    },
    persistence: {
        factory: mosca.persistence.Mongo,
        url: config.DB.mongodb.nodebaseapp.connection
    }
    // secure: {
    //   keyPath: SECURE_KEY,
    //   certPath: SECURE_CERT,
    // }
};

let Mqttsv = function(){
    console.log('Initializing MQTT broker on port 1883');
    server = new mosca.Server(moscaSettings);
    server.on('ready', setup);
    server.on('clientConnected', function(client) {
        console.log('client connected', client.id);
    });

    // fired when a message is received
    server.on('published', function(packet, client) {
        console.log('Published', packet);
        deviceMessageService.insertDeviceMessage('test-client', packet.topic, packet.payloa
    });

    // when client return puback,
    server.on('delivered', function(packet, client){
        console.log('Delivered', packet);
    });
};

/*
 * Module exports
 */
module.exports = new Mqttsv();

Mqttsv.prototype.getServer = function(){
    return server;
};
```

```
function setup() {
    console.log('Mosca server is up and running');
};
```

A.1.12 ConfigIoC.js

```
import awilix from 'awilix';
import router from './api/router';
import * as constants from './utils/constants';
import config from './config/Configuration';

// Import libraries
import express from 'express';
import underscore from 'underscore';
import q from 'q';
import swaggerJSDoc from 'swagger-jsdoc';
import mongoose from 'mongoose';

// Import Utils
import HelpersUtil from './utils/helpersUtil';

// Import Middleware
import CrossOriginMW from './middleware/CrossOriginMW';

// Import controllers
import UserController from './api/user/UserController';
import DeviceController from './api/device/DeviceController';
import DeviceMessageController from './api/device/DeviceMessageController';
import LoginController from './api/user/LoginController';

// Import Factories
import ExceptionFactory from './error/ExceptionFactory';

// Import Services
import UserService from './services/UserService';
import DeviceService from './services/DeviceService';
import DeviceMessageService from './services/DeviceMessageService';

// Import Repositories
import BaseRepository from './repository/BaseRepository';
import UserRepository from './repository/UserRepository';
import DeviceRepository from './repository/DeviceRepository';
import DeviceMessageRepository from './repository/DeviceMessageRepository';

let container = awilix.createContainer({
    resolutionMode: awilix.ResolutionMode.CLASSIC
});

mongoose.Promise = global.Promise;
mongoose.connect(config.DB.mongodb.nodebaseapp.connection, { useMongoClient: true });
```

```
console.log("Initializing Swagger API documentation...");
let swaggerDefinition = {
    info: {
        title: 'RESTful web services for node/express/es6 starter',
        version: '1.0.0',
        description: 'Documentation for node/express/es6 starter',
    },
    host: 'localhost:3001',
    basePath: '/v1/',
    schemes: ['http'],
    consumes: ["application/json"],
    produces: ["application/json"]
};
let options = {
    swaggerDefinition: swaggerDefinition,
    apis: ['dist/api/**/*.js']
};
let swaggerSpec = swaggerJSDoc(options);

console.log("Registering dependencies...");
container.register({
    // Register libraries
    q: awilix.asValue(q),
    express: awilix.asValue(express),
    underscore: awilix.asValue(underscore),
    router: awilix.asClass(router).singleton(),
    config: awilix.asValue(config),
    exceptionFactory: awilix.asClass(ExceptionFactory).singleton(),
    constants: awilix.asValue(constants),
    swaggerSpec: awilix.asValue(swaggerSpec),
    mongoose: awilix.asValue(mongoose),

    // Register middleware
    crossOriginMW: awilix.asClass(CrossOriginMW).singleton(),

    // Register controllers
    userController: awilix.asClass(UserController).singleton(),
    deviceController: awilix.asClass(DeviceController).singleton(),
    deviceMessageController: awilix.asClass(DeviceMessageController).singleton(),
    loginController: awilix.asClass(LoginController).singleton(),

    // Register services
    userService: awilix.asClass(UserService).singleton(),
    deviceService: awilix.asClass(DeviceService).singleton(),
    deviceMessageService: awilix.asClass(DeviceMessageService).singleton(),

    // Register repositories
    baseRepository: awilix.asClass(BaseRepository).singleton(),
    userRepository: awilix.asClass(UserRepository).singleton(),
    deviceRepository: awilix.asClass(DeviceRepository).singleton(),
    deviceMessageRepository: awilix.asClass(DeviceMessageRepository).singleton(),
    // Register utilities
    helpersUtil: awilix.asClass(HelpersUtil).singleton()

});

export default container;
```

A.1.13 server.js

```
import http from 'http';
import * as constants from './utils/constants';
import config from './config/Configuration';
import app from './app';
import Mqttsv from './mqttsv';

const server = http.createServer(app);
const mqttsv = Mqttsv.getServer();

mqttsv.attachHttpServer(server);

let port = process.env.PORT || config.port;
server.listen(port, function () {
    console.log(`Listening on port ${port}`);
});


//Stop process killing on exceptions
process.on('uncaughtException', function (err) {
    console.log('UncaughtException : %s', err.stack ? err.stack : err);
});

server.on('uncaughtException', function (req, res, next, err) {
    console.log('UncaughtException : %s', err.stack ? err.stack : err);
    return res.status(constants.INTERNAL_ERROR).send(err.message);
});

server.on('error', function (err) {
    console.log('Error : %s', err.stack ? err.stack : err);
    switch (err.code) {
        case 'EACCES':
            process.exit(1);
            break;
        case 'EADDRINUSE':
            process.exit(1);
            break;
    }
});
```

A.2 Fabric configuration

A.2.1 org.rdeshapriya.xylem.cto

```
namespace org.rdeshapriya.xylem

participant User identified by userName {
  o String userName
  o String firstName
  o String lastName
}

asset Device identified by deviceId {
  o String deviceId
  o String type
  o String token
}

transaction ChangeDeviceToken {
  o String newToken
  --> Device relatedDevice
}
```

A.2.2 logic.js

```
/**
 * Sample transaction
 * @param {org.rdeshapriya.xylem.ChangeDeviceToken} changeDeviceToken
 * @transaction
 */
function onChangeDeviceToken(changeDeviceToken) {
    var assetRegistry;
    var id = changeDeviceToken.relatedDevice.deviceId;
    return getAssetRegistry('org.rdeshapriya.xylem.Device')
        .then(function(ar) {
            assetRegistry = ar;
            return assetRegistry.get(id);
        })
        .then(function(device) {
            device.token = changeDeviceToken.newToken;
            return assetRegistry.update(device);
        });
}
```

A.3 Xylem dashboard

A.3.1 device-list.component.ts

```
import { Component, OnInit, Inject } from '@angular/core';
import { MatDialog, MatDialogRef, MAT_DIALOG_DATA } from '@angular/material';

import { DeviceService } from '../services/device.service';
import { ProfileService } from '../services/profile.service';
import { Device } from '../data-model/device';
import { DeviceMessage } from '../data-model/deviceMessage';

@Component({
    selector: 'app-device-list',
    templateUrl: './device-list.component.html'
})
export class DeviceListComponent implements OnInit {

    public deviceList: Device[];
    public deviceMessageList: DeviceMessage[];
    public devicesLoaded: boolean;
    public messagesLoaded: boolean;

    displayedColumns = ['topic',  'payload', 'timestamp'];

    constructor(
        private _deviceService: DeviceService,
        private _profileService: ProfileService,
        public dialog: MatDialog
    ) { }

    ngOnInit() {
        this.getDeviceList();
    }

    openCreateDeviceDialog(): void {
        let dialogRef = this.dialog.open(CreateDeviceDialog, {
            width: '350px',
            data: { clientId: '', token: '', type: '' }
        });

        dialogRef.afterClosed().subscribe(result => {
            console.log('The dialog was closed');
            this.getDeviceList();
        });
    }

    getDeviceList() {
        this._deviceService.getDevicesByUser(this._profileService.getCurrentUserId())
            .subscribe(devices => {
                this.deviceList = devices;
                this.devicesLoaded = true;
            }, error => {
                console.log(error);
            });
    }

    viewDeviceMessages(deviceId: string): void {
        this._deviceService.getDeviceMessages(deviceId)
```

```
            .subscribe(deviceMessages => {
                this.deviceMessageList = deviceMessages;
                this.messagesLoaded = true;
            }, error => {
                console.log(error);
            });
    }

}

@Component({
    selector: 'create-device-dialog',
    templateUrl: 'create-device-dialog.html',
})
export class CreateDeviceDialog {

    clientId: string;
    token: string;
    type: string;

    constructor(
        public dialogRef: MatDialogRef<CreateDeviceDialog>,
        private _deviceService: DeviceService,
        private _profileService: ProfileService,
        @Inject(MAT_DIALOG_DATA) public data: any) { }

    close(): void {
        this.dialogRef.close();
    }

    save(): void {
        let ownerId = this._profileService.getCurrentUserId();
        this._deviceService.createDevice(this.clientId, this.type, ownerId, this.token)
            .subscribe(result => {
                console.log(result);
                this.dialogRef.close();
            });

    }

}
```

A.3.2 login.component.ts

```
import { Component, OnInit } from '@angular/core';

import { ProfileService } from '../services/profile.service';
import { User } from '../data-model/user';
import { Router } from '@angular/router';

@Component({
    selector: 'app-login',
    templateUrl: './login.component.html'
})
export class LoginComponent implements OnInit {

    username: string;
    password: string;

    loggedIn: string;

    user: User;

    public storage: Storage = localStorage;

    constructor(private _profileService: ProfileService, private _router: Router) { }

    ngOnInit() {
        this.loggedIn = this.storage.getItem('logged_in');
    }

    login(): void {
        this._profileService.loginUser(this.username, this.password)
            .subscribe(loginResponse => {
                console.log(loginResponse);
                this.user = loginResponse;
                this.storage.setItem('logged_in', 'true');
                this.storage.setItem('user_id', this.user._id);
                this.storage.setItem('username', this.user.username);
                this.storage.setItem('first_name', this.user.firstName);
                this.storage.setItem('last_name', this.user.lastName);
                this._router.navigate(['/device-list']);
            },
            error => {
                console.log(error);
            });
    }

}
```

A.3.3 home.component.ts

```
import { Component, OnInit } from '@angular/core';

@Component({
    selector: 'app-home',
    templateUrl: './home.component.html'
})
export class HomeComponent implements OnInit {

    public storage: Storage = localStorage;
    public loggedIn = 'false';

    constructor() { }
    ngOnInit() {
        this.loggedIn = this.storage.getItem('logged_in');
    }

}
```

A.3.4 device.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/throw';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/do';

import { Device } from '../data-model/device';
import { ServiceConfig } from '../config/config';
import { DeviceMessage } from '../data-model/deviceMessage';

@Injectable()
export class DeviceService {
    constructor(private _http: HttpClient) { }

    getDevicesByUser(userId: string): Observable<Device[]> {
        return this._http.get<Device[]>(ServiceConfig.host + '/users/' + userId + '/devices
            .catch(this.handleError);
    }

    getDeviceMessages(deviceId: string): Observable<DeviceMessage[]> {
        return this._http.get<DeviceMessage[]>(ServiceConfig.host + '/devices/' + deviceId
            .catch(this.handleError);
    }

    createDevice(clientId: string, type: string, ownerId: string, token: string) {
        return this._http.post<Device>(ServiceConfig.host + '/devices', {
            clientId: clientId,
            type: type,
            ownerId: ownerId,
            token: token
        })
            .catch(this.handleError);
    }

    handleError(err: HttpErrorResponse) {
        console.log(err.message);
        return Observable.throw(err.message);
    }
}
```

A.3.5 profile.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/throw';
import 'rxjs/add/operator/catch';
import 'rxjs/add/operator/do';

import { User } from '../data-model/user';
import { ServiceConfig } from '../config/config';

@Injectable()
export class ProfileService {
    public storage: Storage = localStorage;

    constructor(private _http: HttpClient) { }

    loginUser(username: string, password: string): Observable<User> {
        return this._http.post<User>(ServiceConfig.host + '/login', { username: username, pa
            .catch(this.handleError);
    }

    getCurrentUserId(): string {
        return this.storage.getItem('user_id');
    }

    getCurrentUserFullName(): string {
        return this.storage.getItem('first_name') + this.storage.getItem('last_name');
    }

    handleError(err: HttpErrorResponse) {
        console.log(err.message);
        return Observable.throw(err.message);
    }
}
```

A.3.6 app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

import { MaterialDesignModule } from './material-design/material-design.module';

import { ProfileService } from './services/profile.service';
import { DeviceService } from './services/device.service';

import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { LoginComponent } from './login/login.component';
import { DeviceListComponent } from './device-list/device-list.component';
import { AppRoutingModule } from './app-routing.module';
import { CreateDeviceDialog } from './device-list/device-list.component';

@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    LoginComponent,
    DeviceListComponent,
    CreateDeviceDialog
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    MaterialDesignModule,
    AppRoutingModule
  ],
  providers: [
      ProfileService,
      DeviceService
  ],
  entryComponents: [CreateDeviceDialog],
  bootstrap: [AppComponent]
})
export class AppModule { }
```