A Cross-Platform Cloud Middleware for the Internet of Things

Dissertation

Department of Computer Science, University of Leicester
CO7501 Individual Project

Word Count: 16150

Author: Jonathan Sant

Email: js781@student.le.ac.uk

University ID: 149039667

Date: September 2016

Supervisor: Prof. Narayana Jayaram

Second Marker: Prof. Thomas Erlebach

Acknowledgements

I wish to acknowledge the people without whom this would have never been possible. I wish to thank Prof. Narayana Jayaram who has been both a very dedicated tutor and a source of inspiration throughout my dissertation. A big thank you goes to Talitha Dimech for her relentless support throughout this course.

Table of Contents

Table of Figures

Abstract

IoT is confronted with many challenges, yet one of the challenges is inherently predominant. This is the heterogeneity of most existing solutions. There are a multitude of research and industrial projects yet most of them involve either some form of proprietary framework or hardware, or simply constrain the user to one particular system or vendor thus transitioning to a vendor lock-in state.

This project aims to leverage the cloud provider's power without being locked with the provider. This will be done by various means. First, the use of a cross platform technology that is NodeJS. All major cloud providers support this technology and it has an extensive community that is porting libraries from other frameworks every day. A middleware was setup using this technology, together with web standards such as REST APIs and JSON to communicate to the Cloud Controller. This enables a variety of clients and sensor devices to connect, irrelevant of the platform they are running on.

The final and most important feature that made this project a success, is a modular plugin system that wraps around vendor specific logic and provides a common interface for the cloud controller to communicate with the cloud provider's features. This has the added benefit of utilizing the vendor's features without re-inventing the wheel and thus losing features in the process. One example of this would be that most systems provide a messaging hub, with encryption and other security features. The fact that this system will use that same hub makes it secure with the same standards that companies like Microsoft can provide.

The project was evaluated by setting various sensors, registering them with the systems that are setup on various cloud providers, and verifying that the sensors can be remotely invoked and read from.

Chapter 1: Introduction

The greatest innovations in recent history have been those of Cloud Computing and the Internet of Things (IoT). This project aims at creating a system for IoT that uses the benefits that are offered by Cloud Computing i.e. relatively low maintenance costs due to the economies of scale, and virtually unlimited resources.

Although the latter is still a novel concept, there are similar systems that are being developed. However, this project does not simply aim at building such a system, it aims at undertaking one of the most formidable challenges that this paradigm of computing is faced with. IoT incorporates various components; each component contains its own challenges, solutions and technologies. Examples of these are the hardware component, the networking component, messaging component, etc. This wide range of paradigms that IoT encapsulates inevitably creates big challenges that must be overcome for IoT to be a successful and useful technology. This heterogeneous aspect of IoT leads to different providers to limit heterogeneity by creating vendor specific components. While this might alleviate the issue, it creates yet another one; that of vendor lock-in.

This project aims at undertaking this challenge and alleviate this problem. Three levels of abstraction where identified in order to accomplish this: **Operating System Independence, Sensor Communication & Storage** and **Clients Abstraction.** Achieving all these levels would render the system vendor-free.

Operating System abstraction was achieved by choosing a cross-platform programming language. In this case, NodeJS was chosen due to its ubiquity. Client Abstraction was achieved by Implementing a Service Oriented Architecture based Middleware that exposes a REST based API using web standards thus hiding away device communication protocols and data management. Finally, Storage & Communication Protocol independence (between devices and the cloud system) was achieved by making use of the Inversion of Control pattern coupled with the Dependency Injection Pattern. Using these two techniques, the system can operate by executing functions that pertain to an interface. Vendor specific logic could then be written in separate plugins that have to implement these interfaces. This way the system could be configured in different ways on different systems, with different providers, without having to alter the system in anyway. These factors helped avoid vendor lock-in, while taking advantage of an inter-cloud setup.

Finally, a web-based app was packed so that it could be installed on different devices including Android and iOS. This app serves as a Dashboard where users are able to manage their devices remotely.

Chapter 2: Background Literature

There are a number of topic areas for which a literature survey was needed, these topics are as follows:

I. **Cloud Computing**: (Armbrust et al., 2010), (Zhang, Cheng, & Boutaba, 2010), (Pühringer, n.d.)

II. **Internet of Things**: (Ma, 2011), (Pühringer, n.d.), (Doukas & Maglogiannis, 2012)

III. **Combining Cloud Computing and IoT**: (Olawale & Brett, 2016), (Particle, 2016), (Onion, 2016), (Korkmaz et al., 2015), (Botta, de Donato, Persico, & Pescapé, 2014), (Grozev & Buyya, 2014), (Fox, Kamburugamuve, & Hartman, 2012), (Kovatsch, Mayer, & Ostermaier, 2012), (Simmhan, Kumbhare, Cao, & Prasanna, 2011)

IV. **Protocols for IoT**: (Hohpe & Wolf, 2011), (Baronti et al., 2007), ("MQTT V3.1 Protocol Specification," n.d.), (Shelby, Hartke, & Bormann, 2014), (Vinoski, 2006)

V. **Middleware**: (Le Guilly, Olsen, Ravn, Rosenkilde, & Skou, 2013), (Pühringer, n.d.)

VI. **Raspberry Pi as a Sensor Node**: (Vujovi & Maksimovi, 2014), (Maksimović, Vujović, Davidović, Milošević, & Perišić, 2014), (Particle, 2016)

2.1 Cloud Computing

The term "Cloud Computing" has become more and more of a buzz word over the past years, nonetheless it is not just that, it encapsulates one of the greatest innovations in the technological world since the Electrical Grid came along. Similarly, to the electrical grid, cloud computing allows companies and individuals to 'outsource' their in-house computing infrastructure to third party service providers. This allows companies to use computing resources as a service.

Cloud Computing allows companies to outsource both Infrastructure and Software services to large third party companies like Amazon, Google and Microsoft who provide various levels of utility computing. This enables these large companies to build huge data centres across the globe, which enable them to decrease the cost of housing their hardware and subsequently the software that lives on it. This enables providers to allocate more resources, instantly at the whim of the customer or configured to be automated in case there is a sudden peek of requests to the service. This differentiates cloud computing from normal hosting providers or ISPs which usually require some upfront notification to increase machines, memory or storage together with an upfront fixed payment (Armbrust et al., 2010).

As mentioned above, cloud providers are able to deliver various components 'as a service'. The latter is a term that has become practically synonymous with cloud computing. Various levels of computing components are available *as a service*. The most common ones are Hardware as a Service (HaaS), Storage as a Service (STaaS), Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). These levels are stacked upon each other, with each providing a higher level of abstraction.

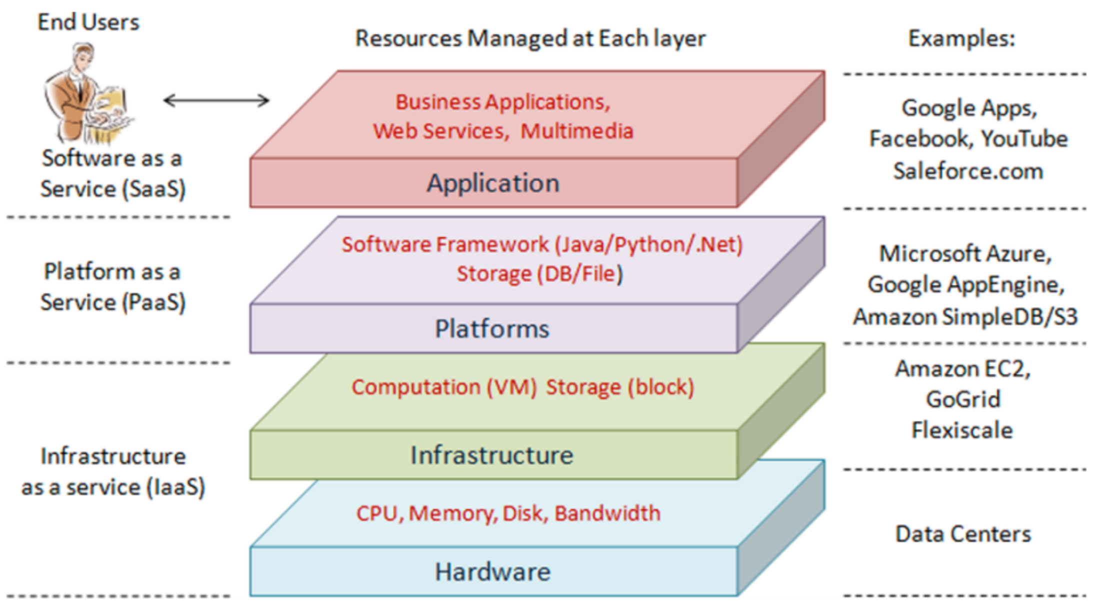

End Users
Software as a Service (SaaS)
Platform as a Service (PaaS)
Infrastructure as a service (IaaS)
Resources Managed at Each layer
Business Applications, Web Services, Multimedia
Application
Software Framework (Java/Python/.Net) Storage (DB/File)
Platforms
Computation (VM) Storage (block)
Infrastructure
CPU, Memory, Disk, Bandwidth
Hardware
Examples:
Google Apps, Facebook, YouTube Saleforce.com
Microsoft Azure, Google AppEngine, Amazon SimpleDB/S3
Amazon EC2, GoGrid Flexiscale
Data Centers

Figure 1 Cloud Layers (Zhang et al., 2010)

2.1.1 IaaS, PaaS and SaaS

The most important concept in cloud computing is the distinction between IaaS, PaaS and SaaS. IaaS is the lowest level of the three (as can be seen in Figure 1). At this level, providers often provide infrastructure i.e. virtual machines managed by some form of hypervisor across various physical machines. Both hardware and networking are abstracted away from the user. However, the user will have to manage the virtual machine, which essentially means that the customer must install servers, software frameworks and platforms and make sure that they are well managed.

On the other hand, PaaS is the pinnacle of cloud computing. This is because it provides the platform upon which application developers may write their applications on. Everything underlying the applications is abstracted away from developers leaving their only concern to be their own application (Pühringer, n.d.). This is incredibly useful, since at this level, businesses can practically thin out their dev ops teams, to the bare minimum and replace these teams with a scalable on-demand infrastructure, costing much less and billed per usage as opposed to upfront costs. This means the infrastructure can support a company from a start-up to huge multi-national corporation with millions of users and transactions per second. However, PaaS comes at the cost of Vendor Lock-in. Using these services requires using provided APIs that are often incompatible with other vendors

(Pühringer, n.d.). Furthermore, the abstraction for the underlying VM might be problematic for advanced uses, which limits the user from modifying it in any way that is not desired by the vendor.

SaaS is the topmost layer in the cloud stack. It is the application built on top of a cloud infrastructure, which provide users with a software system that can be used on demand, without the users having to worry about setting-up and hosting the software. This is convenient for customers since they are billed on demand and can make use of the expertise of the application developers as opposed to trying to build it in-house. As an example we can look at a trading company which is great at making trading software but not an expert on Content Management Systems and hence it can augment its system by making use of a third party CMS API to get its content from (example: (Contentful Team, n.d.)).

2.2 The Internet of Things

The Internet of Things (IoT) is the concept of enabling the interconnection of everyday objects, thus giving rise to the notion of getting status updates from sensors from every aspect in our lives, including dental health from toothbrushes, car stability from car tyres, etc. together with the ability to act upon this data via other '*things*' called actuators.

IoT can be defined as a network that interconnects ordinary physical objects with identifiable addresses in order to provide intelligent services (Ma, 2011). This means that everyday objects become autonomic terminals that are identifiable over the network by a unique identifier and that these devices provide intelligence about the environment that they live in. This adds an extra layer of information to the network and thus making it an *intelligent network* as opposed to the traditional internet, which transmits data and relies on intelligent terminals to decipher that data.

Due to the fact that every object would be transmitting its status and various other sensory information of its surroundings, this data can be gathered and analysed to determine and predict the state of the environment. This permits other actuators to take actions that would improve such environment by determining factors that need to be acted upon, for example energy efficiency of a water heater and the optimal time to turn it on or off, and to actually do so in an autonomous manner.

Machine-To-Machine (M2M) communication is a very broad term that exists in the world of IoT. This simply means some form of technology that is able to communicate over a network to process and act upon the received data (Pühringer, n.d.). The difference of such technology from traditional communication is that in a traditional network, it is the user that initiates some kind of request; in this case however, machines trigger different messages depending on the sensory information that they receive where in turn the receiving end of a triggered message acts upon the data transmitted.

The pinnacle of IoT is with the advent of Smart Cities. The combination of M2M technologies and sufficiently powerful computing power would allow for every aspect in a community or city to be monitored, inefficiencies in some areas identified and acted upon autonomously and proactively (Doukas & Maglogiannis, 2012).

2.3 Cloud Computing and the Internet of Things

Since IoT systems could potentially grow to cover whole cities, being introduced in healthcare, the automotive and other mission critical systems, the IoT must make sure that it is a stable, reliable, secure and highly available infrastructure. This requires an enormous amount of computing power and resources. This aspect of computing, is being tackled by a completely independent branch of the computing industry. This is Cloud Computing. As mentioned in the text above, the Cloud operates in huge data centres across the globe. This gives us virtually unlimited processing power, memory and storage that can be scaled up or down instantly, which are all notions that IoT would greatly benefit from.

It is subsequently natural that IoT and the Cloud are a natural fit. The fact that most cloud providers are introducing IoT into their product family confirms this. AWS IoT was one of the very first systems to show up on a major cloud provider. Amazon has built an IoT infrastructure by taking advantage of its already established cloud system. The system allows a user to create a *'thing'*. A thing can be a sensor or actuator that is registered with the system. Using the SDK provided, on a device, one can register the device with the system using the *things*' credentials. A *thing* can be configured by adding attributes, which are essentially a key value pair. Devices can be configured to update these values and manage attributes.

The architecture to support this infrastructure is made up of three components: The **Control Layer,** controls access to *things*. The **Speed Layer** is the layer that controls communication between things and the cloud system, storing *thing* generated data, and manages rules. The **Serving Layer** manages the web interface that allows users to access the system (Olawale & Brett, 2016).

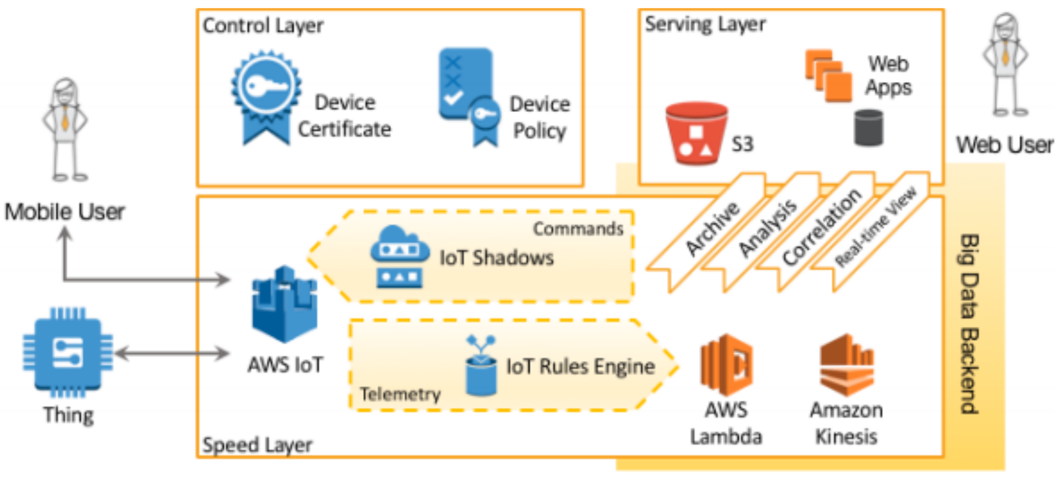


Figure 2 AWS IoT Infrastructure (Olawale & Brett, 2016)

Other cloud providers are also following Amazon's example, Microsoft have added their IoT Hub and similarly Google also introduced IoT into its cloud infrastructure. As of the time of writing the products are still in their early phases with some of the features still in beta or simply not available yet.

Various hardware manufacturers are introducing cloud and IoT. Circuit Board manufacturers are creating boards designed specifically for IoT. The boards come with an SDK that allows the user to connect to a cloud infrastructure, which gives the user the ability to manage all the devices (of the same manufacturer).

Particle Photon is an Arduino like device that costs $19, that has an embedded WiFi and the default firmware installed on it also contains a CoAp server that allows the user to interact with each individual pin on the device via a URL in a RESTful manner (Particle, 2016). A Cloud infrastructure has been setup by Particle (the company the produces the device) that enables the users to command fleets of these devices, enabling over-the-air firmware upgrades or installations. The Photon SDK allows a user to create *Variables and Functions* that act as inputs and outputs to and from the device and the cloud system. This makes it one of the most advanced IoT devices, although similar products like Onion (Onion, 2016) are being created at an even cheaper price (Onion starts a $5).

2.3.1 Heterogeneity & Vendor Lock-in

One of the biggest challenges in the world of cloud computing and that of IoT is the inconceivable amount of heterogeneity amongst various levels in the cloud layer stack. There exist various layers of components that an IoT system, especially one designed to run on the cloud, has to incorporate in order to achieve its purposes. Primarily, a multitude of cloud providers exist and one has to decide which provider to choose. Multiple devices exist that use different hardware and protocols to communicate. Different cloud providers use various operating systems and to complicate things further, operating systems specifically designed for IoT are being introduced. Microsoft have created a version of Windows 10 that can be installed on a Raspberry Pi, specifically designed for IoT (deemed *Windows 10 IoT Core*). Raspberry Pi also comes with its own version of Linux called Raspbian OS. Each platform also comes with its own programming frameworks that a user designing an IoT system must take into consideration. This makes the Heterogeneity challenge that exists in this paradigm of computing a very formidable one indeed (Botta et al., 2014). Each of these vendors and ecosystems usually come with their own programming libraries, design principles, architectural designs, hardware and protocols. This all leads up to transitioning to a vendor lock-in state, where the user is tied to one specific cloud provider using one specific hardware, and one programming framework.

This has various connotations; the first and most obvious one would be the monetary problem. Different providers have different pricing models, while one cloud provider might have cheaper fees on processing power, another one might have cheaper storage fees. The exact same problem exists for suitability of a framework, while NodeJs might be ideal for handling multiple requests and networking applications, Python might be better suited for mathematical operations. Being able to use different providers or frameworks to tackle specific problems is the ideal thing that can be hindered by the vendor lock-in state. Different companies produce different kinds of devices. Not all products can be found under the same company umbrella and thus this leads to either a situation where a particular product cannot be used or a device management problem, since different products will have their own management software.

Some solutions can help with alleviating this problem. One such method is to make use of multiple cloud providers to perform different tasks, and to achieve this, an Inter-cloud architecture must be designed to take advantage of different providers (Grozev & Buyya, 2014). Another way to alleviate the impact is to design the architecture with the necessary levels of abstraction to be able to swap various components and take advantage of varying systems and protocols. This project will try to use a combination of the two, in order to achieve a completely vendor free system.

2.3.2 Cloud IoT Architectures

Architectures for IoT seem to be related. Nearly all authors agree that the system should comprise of four major components (Fox et al., 2012; Korkmaz et al., 2015; Kovatsch, Mayer, & Ostermaier, 2012; Pühringer, n.d.). An **API Controller**, that handles client control, which enable users to manage the devices remotely. A **Messaging Hub** that enables communication from clients to devices and vice versa, and between the devices themselves (M2M). **Clients** that provide the user with a graphical user interface that enable the user to connect with the backend Controller. **Sensors and Actuators** that provide real physical actions to influence their environment or provide data concerning their environment.

(Korkmaz et al., 2015) make use of the whole Google ecosystem. They have used google app engine to host their Web API, Google Cloud Messaging (GCM) to send and receive messages to sensors. The biggest issues with this approach is that the system is very tightly coupled with the Google Architecture. Changes in their SLAs, down time, pricing, etc. will all have an impact on the system. Furthermore, GCM uses XMPP or HTTP, which has two problems. The first is that not all devices support these protocols and hence the authors had to introduce an intermediary hardware controller to coordinate lower powered devices. The second problem is that both of these protocols are resourceful protocols that need more energy to maintain and hence are costlier. (Fox et al., 2012) on the other hand proposes a more flexible architecture and similar to the one that was implemented in this project (see section 3.4). However, the implementation uses specific technologies that would still result in a vendor lock-in architecture.

While also conforming with the general architecture from a high level point of view, (Kovatsch et al., 2012), proposes a very interesting new concept. They propose that the control-flow logic should be created on the cloud. A big component, like a washing machine is split into smaller components, such as *motor, valve, spin-speed.* This enables an entirely shift in mind-set. This way software of various household devices can have features implemented and bugs fixed on the cloud and during the lifetime of the device.

2.3.3 Security & Privacy Concerns

Cloud IoT enjoys the same benefits that any online system possesses. It can be remotely accessible and always available. However, this factor introduced a degree of apprehension from Cloud IoT users. The data gathered by an IoT system can be very personal and potentially dangerous if compromised. Medical records for instance are both very sensitive in nature and potentially dangerous if they get into the wrong hands. Actuators controlling a car remotely can be very dangerous if tempered with.

When an IoT system goes to the cloud, it has essentially been opened to the world. It can be and will be targeted by hackers for some reason or another. (Simmhan et al., 2011) identify different parties and their concerns. They identify three types of actors in the cloud organization: Smart Grid & Utility Providers, Cloud service consumers and third party service providers. Utility providers are usually the providers of power, each having smart meters to measure the power consumption of various areas, households and corporations. They are concerned with data leakage and regulatory compliance since this data can be transmitted to servers all over the world. They are also very concerned about data manipulation. Third Party providers may get concerned with internal algorithms and data that they want to protect. Consumers may be targeted for industrial espionage and SLAs. SLAs can change at the whim of the provider and they might not be compatible with the consumer's requirements, this might be very problematic if the consumer is in a vendor lock-in state. SLA changes might change various factors, including pricing, ownership of the data and data location amongst others. Thus, avoiding vendor lock-in is essential for all parties surrounding the cloud setting.

2.4 Protocols for IoT

While HTTP is a reliable protocol that governs the traditional internet, it is unfortunately not suited for M2M messaging that requires high throughput and availability. HTTP is a text-based protocol that has too much overhead for low powered devices. Furthermore, messages transferred are not binary messages, which greatly increases the file size and efficiency. It is stateless thus a connection must be established every time a message is to be sent/received this also adds to the overhead that comes with HTTP. We need a fast, constrained i.e. resource efficient, yet reliable messaging protocol to be able to communicate with various nodes in real-time. A messaging framework must be able to *Create, Send, Deliver, Receive* and *Process* messages (Hohpe & Wolf, 2011). While many protocols exist, not all protocols are suitable for IoT. ZigBee is a protocol that is usually used for IoT purposes, however it is not considered an IoT protocol since it uses UDP which is a connectionless protocol and hence unreliable (Baronti et al., 2007). ZigBee requires an orchestrator that receives messages over the internet via an IoT protocol and coordinates the ZigBee devices in the WPAN.

2.4.1 Message Queuing Telemetry Transport (MQTT)

MQTT is a constrained protocol that was designed to work with low bandwidth, limited CPU power and memory resources. It is built on top of TCP/IP, which guarantees a reliable connection. This protocol implements the publish/subscribe paradigm and hence a message pattern that provides a one-to-many message distribution. MQTT relies on a central broker, and adds a minor overhead to the message being sent, i.e. a header to ensure a reliable connection and three Qualities of Service: At most once, At least once, and Exactly once. At most once is the lowest QoS, which relies on the reliability of the TCP/IP protocol and hence duplication and loss of messages may occur. At least Once ensures that each message has to arrive at least once, if it does not the message will be published again. However, duplicate messages may occur. The highest QoS is exactly once which ensures that the message is received and that duplication does not occur ("MQTT V3.1 Protocol Specification," n.d.).

2.4.2 Constrained Application Protocol (CoAP)

CoAP is also a constrained reliable protocol and just like MQTT it was designed to work in a very low resource environment. What is special about CoAP is that it was designed to emulate HTTP and more specifically the RESTful protocol. It does not use TCP/IP, it instead uses UDP, which is connectionless and unreliable and therefore has much less overhead and thus it is much more lightweight and fast. However, in order to achieve reliability CoAP has its own mechanisms that make sure that a message is received. It implements the standard verbs used in HTTP i.e. GET, POST and PUT. However, with respect to an M2M environment, data must be pushed from the 'server' (which in this case is just another device in the network) and hence CoAP introduced another verb: OBSERVE, which registers a client so that it can receive push notifications from other devices (Shelby et al., 2014).

2.4.3 Advanced Message Queuing Protocol (AMQP)

AMQP is not usually associated with IoT per se., it is a protocol that was created for situations that demand high reliability and to support enterprise grade communication between sub-systems while decoupling the systems altogether. It is somewhat similar to MQTT however, it supports a lot more features and thus is more suited towards enterprise applications. Microsoft, however, has implemented its IoT hub based on AMQP and this it did with good reason. This is because although it supports many more features than all the IoT protocol out there it uses Binary WIRE that makes it ubiquitous and fast. It supports all the QoS that MQTT supports but also has the concept of queues that ensure that the messaging is reliable. Queues can be durable so that it ensures that even in case of a hardware failure the messages are still safely stored on disk and can be retrieved. It implements all the security features that make it tamper proof and can use different protocols including UDP which can be multicast (just like CoAP) and TCP/IP. It does not only support the pub/sub pattern like MQTT it also supports patterns like asynchronous directed messaging, request/reply and store and forward with all kinds of message routing and forwarding capabilities (Vinoski, 2006). This makes it one of the foremost contenders to be an IoT protocol and Microsoft have done the right move to base their IoT infrastructure upon AMQP.

2.5 Middleware

Middleware plays an essential role in the realm of IoT. Due to the heterogeneous nature of this paradigm it is essential to offer IoT developers the ability to be able to access different technologies via one standardised API (Le Guilly et al., 2013). A middleware is a software system that usually sits in between various systems and provides some kind of API that enables the user of the middleware to interact with different systems that are distributed across different vendors, machines or products. Usually a middleware is implemented as a Service Oriented Architecture (SOA), where a client can be able to communicate with the system without being tightly coupled with it. In IoT the middleware usually hides different communication protocols and subnetworks, which greatly simplifies the client logic since clients no longer have to handle all these different components (Pühringer, n.d.).

2.6 Raspberry Pi as a Sensor Node

(Vujovi & Maksimovi, 2014) and (Maksimović, Vujović, Davidović, Milošević, & Perišić, 2014) describe uses for the Raspberry Pi computer as a sensor node. It describes this board as a small, powerful and a cheap computer. Furthermore, it is designed to waste the least amount of power possible, thus making it a perfect candidate to use for IoT. The Raspberry Pi is a fully-fledged computer running a Linux distribution called *Raspbian OS*. This makes it an extremely flexible and scalable mini-computer and thus it can be used for a large variety of applications. Developers have more freedom and libraries at their disposal including NodeJS, which is a very big benefit towards this project due to the familiarity and strength of the framework. Security becomes less of a concern due to the added power of the board as opposed to other low level controllers were much more effort is needed to implement security measures, if at all possible.

Many boards have started being developed with their primary target being IoT. The Photon (Particle, 2016) has a very powerful ecosystem around it. However, it is not a full-fledged computer like the RPi it is much less powerful with fewer libraries available and with C++ being the only language available. One could argue that since the Photon is cheaper, it is better suited for IoT. Furthermore, it offers inbuilt analogue to digital converter, something that the Pi lacks, although this is something that can be added externally.

Chapter 3: Methodology and Project Contribution

3.1 Problem Description and Statement

Cloud computing is being deemed as the next critical step that will revolutionise the way companies utilize the web. It is said that cloud is what the electrical grid did to the in-house generated electricity. However, there exists a level of apprehension amongst industrial leaders, towards this new paradigm. (Satzger, Hummer, Inzinger, Leitner, & Dustdar, 2013) identify availability of service, data lock-in and legal uncertainties as the main drivers towards this uneasiness. These three problems can effectively be merged into one problem i.e. the vendor lock-in problem. Once some software is written for a certain cloud provider, the software is essentially locked in with the cloud provider. This tends to create dangerous circumstances for the company since it has a dependency on the cloud provider. This can result in loss of data, downtime, and legal issues amongst other problems. This project aims to bring IoT to the cloud. IoT also adds to the problem of vendor lock-in (Botta et al., 2014). IoT adds physical devices and communication protocols to the aforementioned problems.

3.1.1 Levels of Abstraction

To achieve a true cross-platform solution in the context of both IoT and Cloud Computing there are varying levels that one must consider to achieve the right solution. The first level that must be addressed is *Operating System Independence*. This is the most crucial level to address since different cloud providers use different operating systems, some of them even implementing their own version of OS (Garfinkel, 2007) (although these are usually based on a Linux kernel).

The next level is the *Communication Protocol Abstraction* level. There exist a multitude of protocols that are used in the world of IoT (Mineraud, Mazhelis, Su, & Tarkoma, 2016) and one needs to make sure that this level is abstracted to be able support multiple devices that use different protocols to communicate.

The system needs to register its devices, device and user information. However, not every cloud provider uses the same storage systems. Furthermore, the owner of the system might decide to store the data privately on private servers since the data might be sensitive. Hence, a *Storage Abstraction* is also required.

Finally, the system should be able to serve any client application that wishes to control and read the statuses off these devices.

Level	Abstraction Components
1	Clients
2	Sensor Communication & Storage
3	Operating System Independence

Table 1

3.2 Approach, Tools and Techniques

3.2.1 Approach

In section 3.1.1 it was determined that in order to achieve the goal of a true cross-platform[1] IoT system the three levels of abstraction mentioned should be tackled. Most cloud services providers nowadays offer many services that can be used in IoT. Services like storage, a messaging bus, application hosting, etc. Most of these providers have also started to provide IoT specific infrastructures (Microsoft, 2016). The infrastructure proposed in this project should be able to take advantage of the features provided by the cloud providers, which is inherently obvious that they enjoy vaster recourses and can provide a very robust and efficient architecture.

The role of the IoT system proposed in this project is thus that of a Middleware. A middleware provides a set of custom functionality on top of the operating system that it resides on, to connect various systems, software services and components together to achieve some preconceived business task (Bernstein, 1996). Hence, in this case, a business or user can still leverage the cloud provider's expertise and infrastructure.

[1] Cross-platform, in the context of this project, means to have the ability to be completely abstracted away from the actual components of the infrastructure it is deployed on.

The latter is the first role that the IoT system or *middleware* has to play. The second role is to abstract away the communication protocols that are used to communicate with the devices. This can be achieved by using web standards that are used nowadays. To achieve this requirement, the IoT middleware must be implemented in the form of a REST API. This makes the system available to a plethora of clients. A client can be a mobile, desktop or web application that has the ability to communicate via HTTP. This means that the client does not need to know or use any other communication protocol, thus resolving the first level of abstraction between clients and communication protocols.

To resolve the third level of abstraction is perhaps the simplest of all the other levels. This requires a cross-platform programming language. There exist various languages of such nature, however one must make the correct decision.

Finally, a dashboard client must be developed so that a GUI interface is provided to the user to grant the ability to monitor and control the sensor devices.

3.2.2 Tools

As stated above, the simplest level to achieve is the Operating System abstraction. A suitable programming language must be determined. It needs to be able to run on multiple operating systems and it must be supported by most of the major cloud providers. There are various languages that can accomplish this task, the most predominant are Java and Node.js.

NodeJS was chosen to be the programming language of choice for a number of reasons. Unlike other languages and frameworks, all the major cloud vendors accept NodeJS. Javascript is becoming more ubiquitous, finding its way to web applications, desktop applications, and mobile applications. It can be used for both client side programming and server side programming. NodeJS also has a very small footprint and thus it can run on lower power devices like the Raspberry Pi. Furthermore, libraries that exist in other frameworks are being added to the NodeJS ecosystem, thus providing an immense library of plugins and libraries. NodeJS is built to be very efficient at handling multiple I/O requests and was found to outperform many of established languages and frameworks (Chaniotis, Kyriakou, & Tselikas, 2015). This makes very suitable to take on the role of an IoT Middleware. On top of Node, Express.js in order to facilitate the creation of REST APIs and common HTTP tasks.

The drawback of using NodeJS is that it uses an incomplete language i.e. Javascript. Javascript was built to be a scripting language that is used for simple web tasks. Its un-typed nature, which makes it easy to build quick prototypes, becomes a huge unmaintainable mess, as soon as the project grows beyond a certain amount of LOC. For this reason, the system is being build using Microsoft's TypeScript(TS). TS is a superset of Javascript that allows the programmer to use EcmaScript2016 features, which include superior inheritance and class mechanisms which are basic OOP features, amongst other features that significantly improve Javascript. TypeScript also adds type definitions. This way the programmer can use static analysis to navigate around the project with ease and detect potential errors in advance. This makes Javascript much more maintainable and thus why it is suitable for this kind of project (Bierman G, Abadi M, 2014).

There are other tools that have been used in this project, however, in this section only those that help resolve the problem that this project aims to resolve have been highlighted in this section.

3.2.3 Techniques

The fulcrum of this entire project is to be able to abstract away as much as possible, the underlying platform and any third party services that are required for this project to work, most importantly storage services and the messaging bus. This will enable the system to achieve the abstraction required at level two in Table 1.

To achieve this, a plugin concept has been conceived, where any external entity[2] is a module that can be installed and interchanged with an equivalent module. A plugin provides vendor specific logic that can operate a cloud component, example: a database. This means that to change from one vendor to the next what is required is simply to install the plugin of the vendor that is required.

Being able to do this without requiring to change the core logic of the system, whenever a new plugin needs to be installed requires a modular architecture. This is achieved via one specific design pattern. This pattern is *Dependency Injection (DI)* and *Inversion of Control (IoC).* Dependency Injection or more specifically Interface injection, allows an engineer to write the code against an interface instead of a concrete implementation. The concrete implementation would only be injected at runtime through the constructor (Martin Fowler, 2004). In order to write a new plugin, in this case, all that would be needed is to implement

[2] For the scope of this project we are only considering Storage and Messaging as external entities

the provided interfaces and hence no core logic needs to be changed since the same interface is still being used.

```
constructor(
    private variableService: IVariableService,
    private sensorService: ISensorService,
    private streamService: IStreamService,
    private messagingService: IMessagingService,
    private logger: Logger,
    private contextService: IContextService) {

}
```

Figure 3: Constructor injection

Figure 3 is the constructor of the *VariableProvider* class that uses the specified interfaces without any knowledge of the actual implementation (types denoted with an '*I*' are interfaces, a convention picked up from the C# programming language).

3.3 Requirements

The requirement specifications-functional, non-functional and domain specification categories are as follows:

ID	003
FUNCTION	Sensor Registration
TYPE	Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Sensor API
DESCRIPTION	A user should be able to register multiple a Sensors and associate it with a home. A sensor should specify an alias and protocol it intends to use, which will be used to generate messages using the appropriate protocol. Upon registering a sensor, the system shall generate a UUID that shall be used by the sensor to identify itself. Along with the sensor itself, the user should be able to register *variables*[3] and *jobs*[4]. A *variable* should have a name, a data type and a flag that determines whether it is a stream or not. A *Job* requires a name, and input parameters.
FITNESS CRITERION	A REST endpoint under the resource */Sensor/Register* should be created. Users can create POST requests with the aforementioned data in request payload. A new Sensor should be added on the data base together with its *variables* and *jobs*
DEPENDS ON	011

[3] A *variable* in this case is an attribute of the sensor, example: temperature, on/off, etc.
[4] A *job* is a remotely invokable feature of the sensor that makes the sensor perform some action.

ID	004
FUNCTION	Retrieve Sensors
TYPE	Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Sensor API
DESCRIPTION	A user should be able to retrieve a list of sensors.
FITNESS CRITERION	A REST endpoint under the resource */Sensor/List* should be created. Users can create GET requests to retrieve a list of objects (representing sensors).
DEPENDS ON	011

ID	006
FUNCTION	Retrieve Jobs
TYPE	Functional
PRIORITY	MUST

DESTINATION	IoT Cloud Controller/Sensor API
DESCRIPTION	This should yield a list of *jobs* pertaining to a particular sensor.
FITNESS CRITERION	A REST endpoint under the URL */Sensor/Remote {sensorId}* should be created. Users can create GET requests to retrieve a list of *job* objects together with their parameters.
DEPENDS ON	012

ID	007
FUNCTION	Invoking Jobs
TYPE	Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Sensor API
DESCRIPTION	A user can invoke a job that will make the sensor perform some action.
FITNESS CRITERION	A REST endpoint under the URL */Sensor/Remote/{sensorId}/{jobName}* should be created. Users can create POST requests, supply the parameter data required by the job. This should trigger a message, which will be sent to the sensor.
DEPENDS ON	

ID	008
FUNCTION	Plugin Support
TYPE	Non-Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Plugin SDK
DESCRIPTION	The system should be modular and coupling between modules should be minimal. Modules should always be injected into other modules as dependencies. Modules should be programmed against interfaces of the modules they depend on. This will ensure customizability without compromising the system, makes it deployable to multiple environments.
FITNESS CRITERION	The system should keep operating normally regardless of which plugin it is using (provided that the plugin works correctly on the current environment).
DEPENDS ON	

ID	009
FUNCTION	User Persistence
TYPE	Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Storage API
DESCRIPTION	CRUD operations for managing users
FITNESS CRITERION	The system must be able to create, retrieve update and delete user data on cloud storage.
DEPENDS ON	013, 014

ID	011
FUNCTION	Sensor Persistence
TYPE	Functional
PRIORITY	COULD
DESTINATION	IoT Cloud Controller/Storage API
DESCRIPTION	CRUD operations for managing Smart Areas

FITNESS CRITERION	The system must be able to create, retrieve update and delete smart area data on cloud storage.
DEPENDS ON	013, 014

ID	012
FUNCTION	Sensor Persistence
TYPE	Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Storage API
DESCRIPTION	CRUD operations for managing Sensors and ability to handle streams of data
FITNESS CRITERION	The system must be able to create, retrieve update and delete sensor data on cloud storage.
DEPENDS ON	013, 014

ID	013
FUNCTION	MongoDB Storage Plugin

TYPE	Domain
PRIORITY	SHOULD
DESTINATION	IoT Cloud Controller/Plugin SDK
DESCRIPTION	A plugin that provides the storage capabilities required by this system and abstracts the integration to MongoDB
FITNESS CRITERION	CRUD operations can be performed on MongoDB.
DEPENDS ON	013, 014

ID	015
FUNCTION	Messaging API
TYPE	Non-Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Messaging API
DESCRIPTION	*Variables* and *jobs* are to be used as interfaces that abstract away the actual communication layer from the system. *Variables* should serve as interfaces for output messages, while *jobs* should serve as interfaces for input messages.

FITNESS CRITERION	The system should be able to send messages to different sensors and receive messages.
DEPENDS ON	016, 017, 018

ID	017
FUNCTION	AMQP Plugin
TYPE	Functional
PRIORITY	SHOULD
DESTINATION	IoT Cloud Controller/Messaging SDK
DESCRIPTION	A plugin that encapsulates communication with AMQP Hub.
FITNESS CRITERION	The system should be able to send messages and receive messages sent through AMQP (RabbitMQ)
DEPENDS ON	008

ID	018
FUNCTION	MQTT Plugin
TYPE	Domain

PRIORITY	SHOULD
DESTINATION	IoT Cloud Controller/Messaging SDK
DESCRIPTION	A plugin that encapsulates communication with MQTT message broker.
FITNESS CRITERION	The system should be able to send messages and receive messages sent through MQTT
DEPENDS ON	008

ID	024
FUNCTION	Sensor Registration Screen
TYPE	Functional
PRIORITY	MUST
DESTINATION	Dashboard/Sensor/Sensor Maintenance
DESCRIPTION	A screen should be presented to the authorised users where they can add sensors, by supplying the alias, the protocol that the sensor uses and the Smart Area to be associated with. This will post the sensor to the backend service and will be added to a list on the same screen. The screen shall display a list of sensors upon screen load. Each sensor listing should display its alias and UUID generated.
FITNESS CRITERION	The client application should send a POST request on the provided endpoint with the aforementioned data in the requests payload to register the sensor. The sensor created should be displayed in a list on the same screen.
DEPENDS ON	003

ID	025
FUNCTION	*Variable* Registration Screen
TYPE	Functional

PRIORITY	MUST
DESTINATION	Dashboard/Sensor/Sensor Maintenance
DESCRIPTION	Each listing in the sensor registration screen shall display a button that should trigger a screen to register *variables*. The *variable* registration screen shall prompt the user to supply the name, data type and a whether the variable is a stream or not.
FITNESS CRITERION	The client application should send a POST request on the provided endpoint with the aforementioned data in the requests payload to register the a *variable*.
DEPENDS ON	003

ID	026
FUNCTION	*Job* Registration Screen
TYPE	Functional
PRIORITY	MUST
DESTINATION	Dashboard/Sensor/Sensor Maintenance
DESCRIPTION	Each listing in the sensor registration screen shall display a button that should trigger a screen to register *jobs*. This screen

	shall prompt the user to supply the name and parameters it should be supplied with.
FITNESS CRITERION	The client application should send a POST request on the provided endpoint with the aforementioned data in the requests payload to register the *job*.
DEPENDS ON	003

ID	027
FUNCTION	*Variable* Visualisation Screen
TYPE	Functional
PRIORITY	MUST
DESTINATION	Dashboard/Sensor/Variable
DESCRIPTION	Each variable shall have its own screen so that it can display its current status and a visualisation of the historical data.
FITNESS CRITERION	The client application should send a GET request on the provided endpoint with the *variable* ID to retrieve the variable data.
DEPENDS ON	015

ID	028

FUNCTION	*Job* Invocation Screen
TYPE	Functional
PRIORITY	MUST
DESTINATION	Dashboard/Sensor/Jobs
DESCRIPTION	Each job shall have its own screen where the user can supply the data for each parameter and request the cloud controller to send a message to the sensor.
FITNESS CRITERION	The client application should send a POST request on the provided endpoint with the parameter data ID to the cloud controller.
DEPENDS ON	015

ID	030
FUNCTION	Cross Platform Dashboard Application
TYPE	Non-Functional
PRIORITY	SHOULD
DESTINATION	Dashboard/Rule
DESCRIPTION	The dashboard application should be cross platform. It should be built using a cross platform technology, where the application shall be ported to each platform with minimal effort.
FITNESS CRITERION	The dashboard application should work on at least Web, Android and iOS.
DEPENDS ON	

3.4 System Design

A system consisting of four main components was designed in order to satisfy the requirements, mentioned in section 3.2, and to overcome the challenge that this project set out to tackle, mentioned in sections 3.1 and 3.1.1. These components are indigenous to most IoT projects, as clearly stipulated in (Fox et al., 2012) and (Microsoft, 2016). These four components are defined as follows for the purpose of this project:

- Controller
- Messaging Broker
- Clients
- Sensor Nodes

3.4.1 Cloud Controller

The Cloud Controller is the core of the system, the middleware that sits in between the cloud system, platform configuration, clients and sensors. It abstracts away the interaction with the cloud system, thus making any setup work with whichever provider is deemed fit. Furthermore, the architecture abstracts away any storage and communication integration so that as many as possible devices and technologies are compatible. This will be achieved by various methodologies.

Camp IoT, which is the name of this system, encapsulates all three levels of abstraction mentioned in section 3.2.1. Camp IoT is implemented in the form of an HTTP REST API. This makes the system standards compliant and available to virtually any client. As opposed to (Fox et al., 2012) the Broker component, which will be explained in further detail below, is not visible to the client thus enabling the client to communicate via standard methods, relieving concern with which protocol to communicate (and thus free of vendor lock-in) and relying on one simple implementation as opposed to a myriad of protocols.

The controller was written in Javascript/NodeJS. Unlike other languages and frameworks NodeJS is accepted by all the major cloud vendors. JavaScript is becoming more ubiquitous, finding its way to web applications, desktop applications, mobile applications and low powered boards like the Raspberry PI. It can be used for both client side programming and server side programming. Furthermore, libraries that exist in other frameworks are being added to the NodeJS ecosystem, thus providing an immense library of plugins and libraries.

3.4.2 Messaging Broker

The messaging broker is very important component to the system. It handles communication between the devices and the controller as well as between devices themselves. However, in order to achieve the goals of this project, the broker should not be part of our architecture. Each cloud provider has some kind of messaging bus. Other messaging broker service providers also put their service on the cloud. It is also possible to put a custom setup on a cloud VM. This means that there are plenty of options for one to choose from and hence integration of the controller to the broker has to be via a plugin that can be interchanges with different brokers or protocols.

In this project, **RabbitMQ** was chosen as the messaging broker. This message broker is special because it is free, cross-platform, it has its own cloud infrastructure (if the user doesn't want to set it up manually) and supports various protocols, including AMQP, MQTT, STOMP and HTTP (Videla & Williams, 2012). This makes RabbitMQ ideal since it helped with testing different protocols easily without too much setup and without incurring too many costs.

Though very versatile, a user might still get constrained with RabbitMQ, because it is strictly useful for devices that support the above protocols. There are other protocols that are used in the IoT world, one of which is CoAp. This protocol does not use a message queue, and hence is not supported by RabbitMQ. This makes it even more clear why any form of communication with the sensors needs to be done through a plugin and abstracted in the controller code.

3.4.3 Dashboard Clients

The clients in this project are software systems that enable users to connect to the cloud system, get status information about the sensors, manage sensors and remotely invoke *jobs* that the sensors expose.

The primary medium that the dashboard should be delivered to is the web. The fact that the web is ubiquitous, makes it inherently multi-platform. This is achieved via standard web technologies including HTML5, CSS3 and JavaScript, which will be enhanced by various frameworks including SCSS and AngularJS and Typescript.

While the web medium would suffice for the success of this project, users tend to prefer native apps on their phones. However, to reduce the impact of having to implement the system at least three times, technologies like Foundation for Apps and Cordova were used. This enabled the web app to be available natively on iOS and Android with minimal changes it while also gaining access to the native features of the phone, like location and other sensor information.

3.4.4 Sensor Nodes

While there is a myriad of devices that can connect to the system, the primary sensor device that will was used in this project is the Raspberry Pi. This is because the RPi is a very powerful system (Vujovi & Maksimovi, 2014) and thus gives it the ability to be flexible, and able to support multiple technologies with reduced effort and thus focus can be given to the cloud controller which is the focus of this project. The Raspberry Pi made it possible to use the same language that both the Controller and Dashboard were written in i.e. JavaScript and NodeJS and thus reducing the effort needed to demonstrate the capabilities of this system.

The Sensor Nodes do not communicate directly to the controller, instead they communicate directly to the broker. This is because the device should know what protocol it wants to use and should be able to access all the broker's features.

3.4.5 Architecture

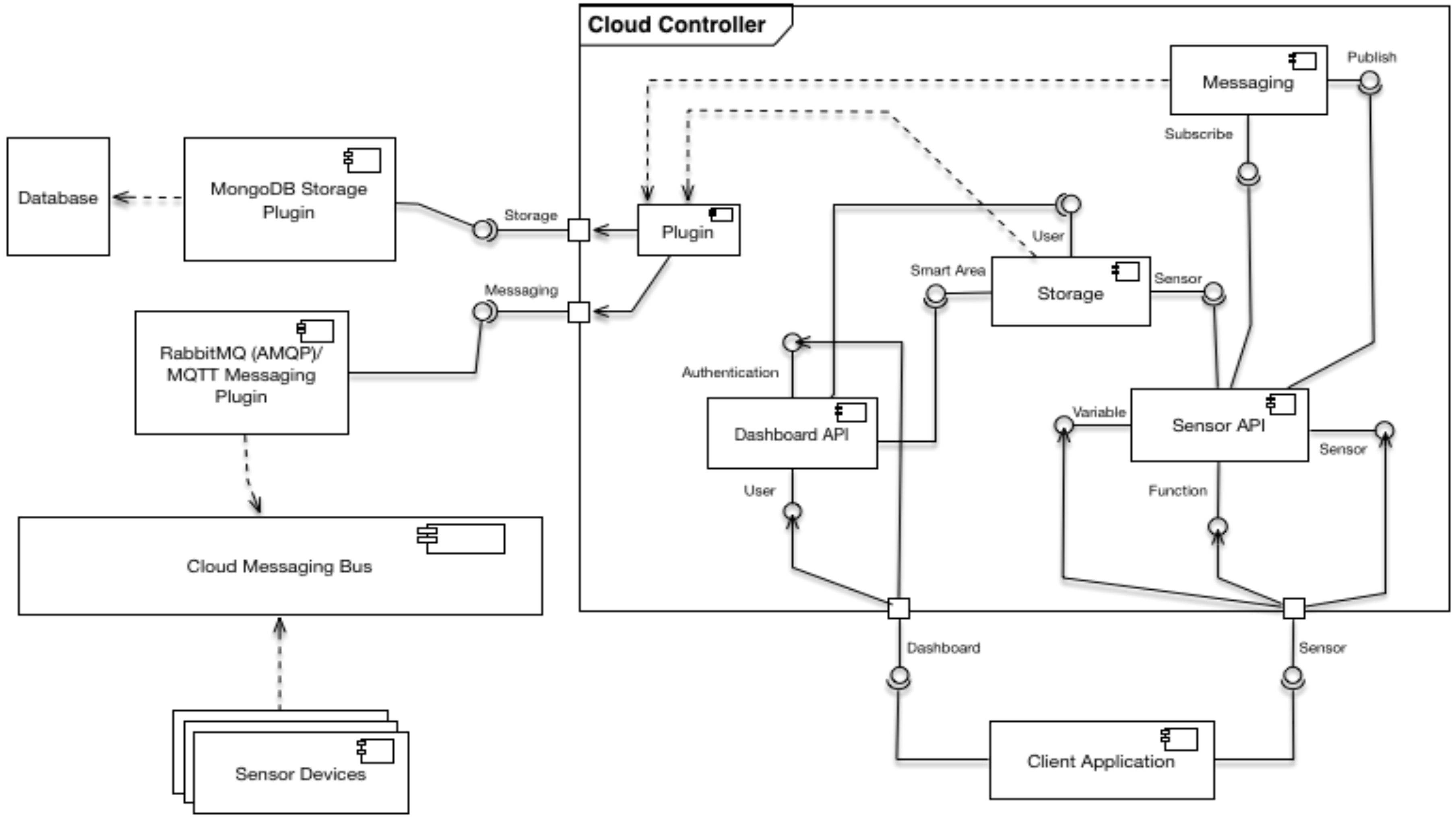


Figure 4

3.5 Implementation

In this section the implementation of the system will be broken down into various components in order to go into the necessary details to explain how the design in section 3.4 was achieved. This section will start by illustrating the project structure of the project to the deployment phase.

3.5.1 Versioning

As with any modern software system a version control system was used in order to have a code backup on a remote server and most importantly to be able to track versions and record history of changes, in case something goes wrong and the need to pinpoint the exact time were the problem was created, can be identified.

GIT was used since it is the most predominant version and modern continuous delivery techniques in use nowadays revolve around it (Krusche & Alperowitz, 2014a). More importantly, the *gitflow* workflow or branching model that exists within the GIT ecosystem was employed. (Krusche & Alperowitz, 2014a) explain the git-flow in depth, for the context of this project, new features where always developed in their separate 'feature' branches and merged to the develop branch whenever they were deemed ready. Whenever a deliverable was required (for instance for Milestones 1 and 2) a release branch would be created from develop, then merged to the master branch and tagged with the version number. This flow decreases the probability of erroneously releasing a broken build, since the develop and master branches are not pushed directly to unless the appropriate testing has been carried out. This means that the develop and master branches are always working version of the system.

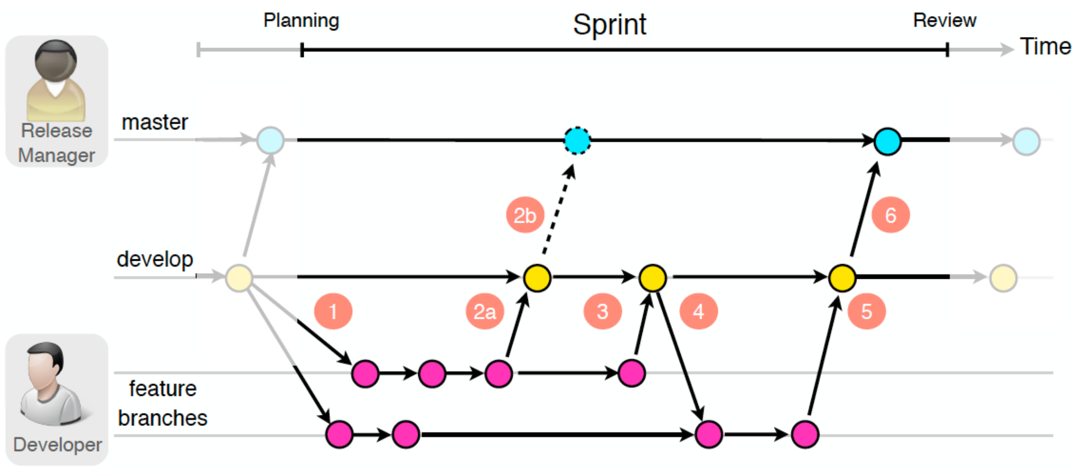


Figure 5 Gitflow as illustrated by (Krusche & Alperowitz, 2014b)

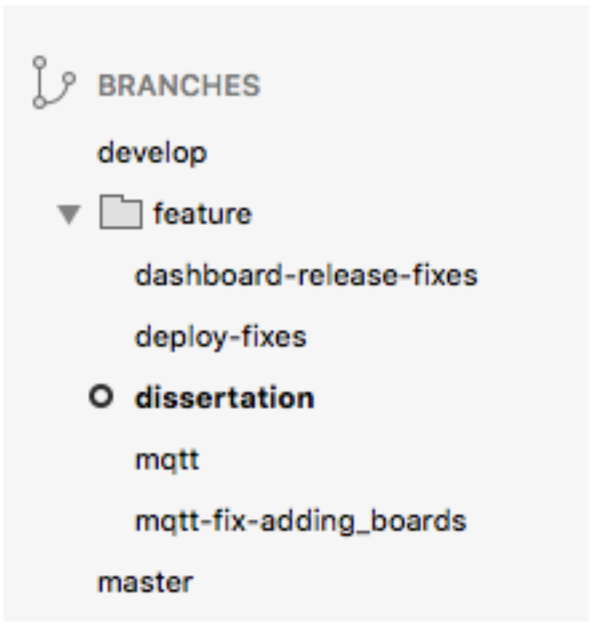
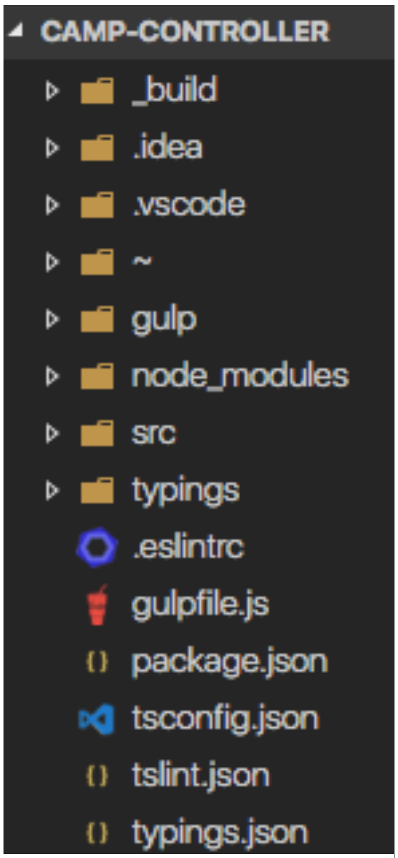

BRANCHES
develop
feature
dashboard-release-fixes
deploy-fixes
dissertation
mqtt
mqtt-fix-adding_boards
master

Figure 6The actual branches in SourceTree

3.5.2 Project Structure

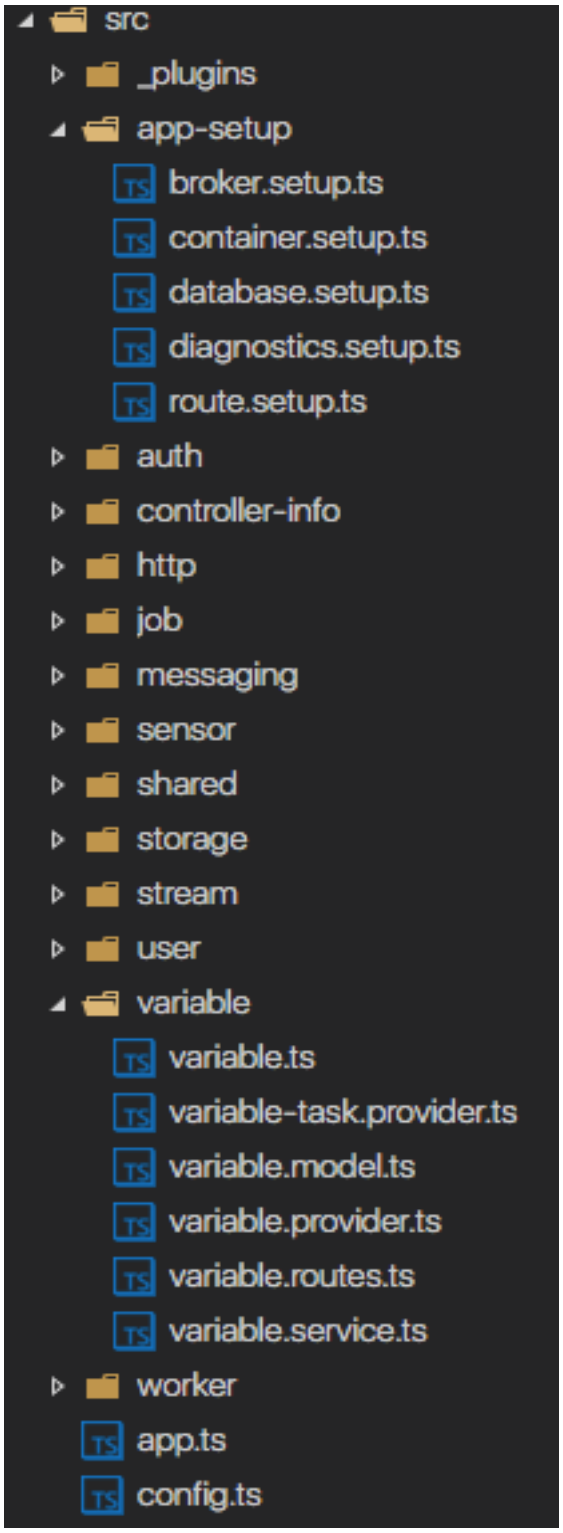
Though it may seem insignificant the project structure can contribute a lot, to how the cleanly the code is structured and how easily one can determine how it all fits together. It is for this reason that this section is being included.



```
CAMP-CONTROLLER
  _build
  .idea
  .vscode
  ~
  gulp
  node_modules
  src
  typings
  .eslintrc
  gulpfile.js
  package.json
  tsconfig.json
  tslint.json
  typings.json
```

Figure 7 Project Structure: Root Files/Folders

Figure 3Figure 7 above shoes the root folders. The most important directories and files out of all of these are the 'gulp', 'src' and 'typings' directories and the 'gulpfile.js' file. Gulp is a task runner built on top of NodeJS, more details are specified in section 3.5.3. The 'gulp' directory contains the gulp tasks that help build the project, run tests and deploy the project. The 'src' directory contains the actual source code. The 'typings' directory contains the definition files that are required by Typescript in order to get statically types information on any third party Javascript libraries that the project uses. These definition files are either manually defined or provided by the third party libraries creators (or built by the community). The 'gulpfile.js' file, is required by Gulp as the entry point to start running the tasks.



```
src
  _plugins
  app-setup
    broker.setup.ts
    container.setup.ts
    database.setup.ts
    diagnostics.setup.ts
    route.setup.ts
  auth
  controller-info
  http
  job
  messaging
  sensor
  shared
  storage
  stream
  user
  variable
    variable.ts
    variable-task.provider.ts
    variable.model.ts
    variable.provider.ts
    variable.routes.ts
    variable.service.ts
  worker
  app.ts
  config.ts
```

Figure 8 The Src directory

The most important directory is the 'src' directory since this contains all the source code that makes the system work. Figure 8 shows the structure of this directory. Each component in the system resides in its own directory and all items related to a particular directory should be found in the same directory. This kind of classification by 'domain' help make it easier to find what one might be looking for while also make it easier for modularization. Which means that if one component must be used in some other system, it can be easily extracted into its own NPM module and shared between both projects. The latter is also both encouraged and at the heart of the NodeJS ecosystem. Note also the fact that each file is denoted by the module/component name followed by what functionality that file provides ex: `variable.service.ts,` where variable is the component that handles variables (Camp Variables that can be found in the requirements section 3.3).

The directory named '_plugins' is a directory that contains the plugins code i.e. the platform specific logic that integrates the controller to the Cloud System's components. The directory is denoted by an underscore to make it visible that the contents of this folder should not reside within this project. They should instead reside in their own respective repositories, packaged as NPM packages and installed within this project. The reason that they are included in this project is for ease of development, since these plugins were being developed alongside the rest of the controller, it made it easier to edit these plugins as opposed to repackaging and reinstalling every time a code change needed to be affected.

The file 'config.ts' contains hard coded connection endpoints and settings that are constant configuration items, required for the operation of the system while 'app.ts' is the main entry point of the system. This file is the first file that is executed and bootstraps the system.

3.5.3 Gulp and Project Build

Gulp is a task/build runner, built using NodeJS. It is being used not just Node and JavaScript projects but by anyone who wishes to automate their builds, deployments, etc. Since Typescript is being used in this project, Node cannot simply execute .ts files. The V8 engine, which Node is built upon, is a JavaScript engine that executes JavaScript. Typescript is merely a language to write typed JavaScript but is not the actual implementation and hence this needs to be transpiled into JavaScript. This can be done manually by transpiling each file individually, however, in practice something more automated is required. This is the main reason that Gulp is being used.

In the gulp directory we can find three[5] main task files, each corresponding to a particular task (following the same structure that the src exhibits). The *build* task handles retrieving all the .ts files in the src folder, passing them through a Typescript linter to check for any errors and finally transpiling them to Javascript and outputting them in the '_build' directory. The clean task is a simple task that cleans the '_build' and deploy directories. The tests task retrieves all .spec.ts files, builds them, executes the unit tests that they represent and output a report to highlight the status of the tests.

```
const gulp = require("gulp");
const tslint = require("gulp-tslint");
const ts = require("gulp-typescript");
const stylish = require("gulp-tslint-stylish");
const sourcemaps = require("gulp-sourcemaps");
const gIf = require("gulp-if");

exports.build = (globList, produceSourceMaps, tsOptions) =>
{
    var stream = gulp.src(globList)
    .pipe(gIf(produceSourceMaps, sourcemaps.init()))
    .pipe(tslint({
    configuration: "tslint.json"
    }))
    .pipe(tslint.report(stylish, {
    emitError: false,
    sort: true,
    bell: true
    }))
    .pipe(ts(tsOptions, undefined,
ts.reporter.fullReporter(ts.reporter.fullReporter(true))))
    .pipe(gIf(produceSourceMaps, sourcemaps.write()));

    return stream;
};
```

Figure 9 The build task (in Javascript (ECMAScript2016))

[5] There are actually more tasks but the three highlighted are the main tasks, all other tasks are some form of aggregation of these tasks

3.5.4 Inversion of Control and Plugins

As stated in section 3.2.3 the plugin mechanism will be made possible with the Inversion of Control and Dependency Injection techniques. These two concepts have become almost standard patterns to use in any modern architecture. However, in JavaScript this concept is somewhat alien. This is because JavaScript is a loosely typed language and the need for dependency injection is required less. The case for modularity is, nevertheless, still prevalent and though you can achieve some form of DI in JavaScript, having a structured way of doing so is desirable. This project also employs the use of typescript, which turns the loosely typed nature of JS into a strongly typed one. Although TS makes it possible to write plain JS, it would make it an anti-pattern, rendering the use of TS useless.

Attempts at dependency injection in various JS frameworks do exists (Knol, 2013) and this projects uses some of the existing architecture design to implement a custom IoC container has been implemented, together with a standard method of defining dependencies in classes in order to be injected.

In order to create a DI mechanism successfully, the container must implement the following features:

- Ability to specify which concrete implementation should correspond to a given interface.
- Ability to create a new object provided for a given type (or interface type).
- Ability to identify the class' dependencies and resolve them.
- Ability to initialize the dependencies and their dependencies recursively.
- Ability to specify whether a dependency should be a Singleton, throughout the lifecycle or created every time it is requested.

The above implementation takes form through these three parts:

- `ContainerService` class
- `ContainerItem` class

- The `$inject` array specifying the dependencies in the classes requiring other classes (dependencies).

The `ContainerService` class is essentially a `HashMap` that uses the type as a key and a `ContainerItem` as a value. This class has some publically available functions to be able to create a class while resolving its dependencies and to provide a singleton instance quickly by invoking the `HashMap.`

```
export class ContainerService {
    private container = new Map<string, ContainerItem>();

    register(item: ContainerItem): void {
        if (!this.container.has(item.key)) {
            this.container.set(item.key, item);
        }
    }

    getByKey<T>(key: string): T {
        let item = this.tryGet(key);
        if (this.shouldInstantiate(item)) {
            throw `${key} is not a singleton and it should
be created. Try using the create function instead.`;
        }

        return item.item;
    }

    create<T>(instantiatee: {new(...args: any[]): T; }): T {
        let injections = (<any>instantiatee).$inject;
        let items: any[];
        if (injections) {
            items = this.resolve(injections);
        }

        const _instantiatee = instantiatee;
        let ins = function (...args: any[]) {
            return new _instantiatee(...args);
        };
        ins.prototype = _instantiatee;

        ins = ins.apply(null, items);

        return <T><any>ins;
    }
}
```

Figure 10 The ContainerService class (private methods have been omitted for simplicity's sake)

Figure 10 above is a snippet of the `ConatinerService` class and its public functions. The `register` function is a simple function that takes a `ContainerItem` and adds it to the internal `HashMap`.

The `getByKey` function returns the correct class instance for a given key, with the key being an interface identifier. This function only works for singleton items, since other classes need to be instantiated every time they are requested and since instantiating a class requires resolving all the dependency tree.

The latter needs to be done by invoking the function `create`. This is what triggers the dependency resolving mechanism that is highlighted in the form of a flow chart below:

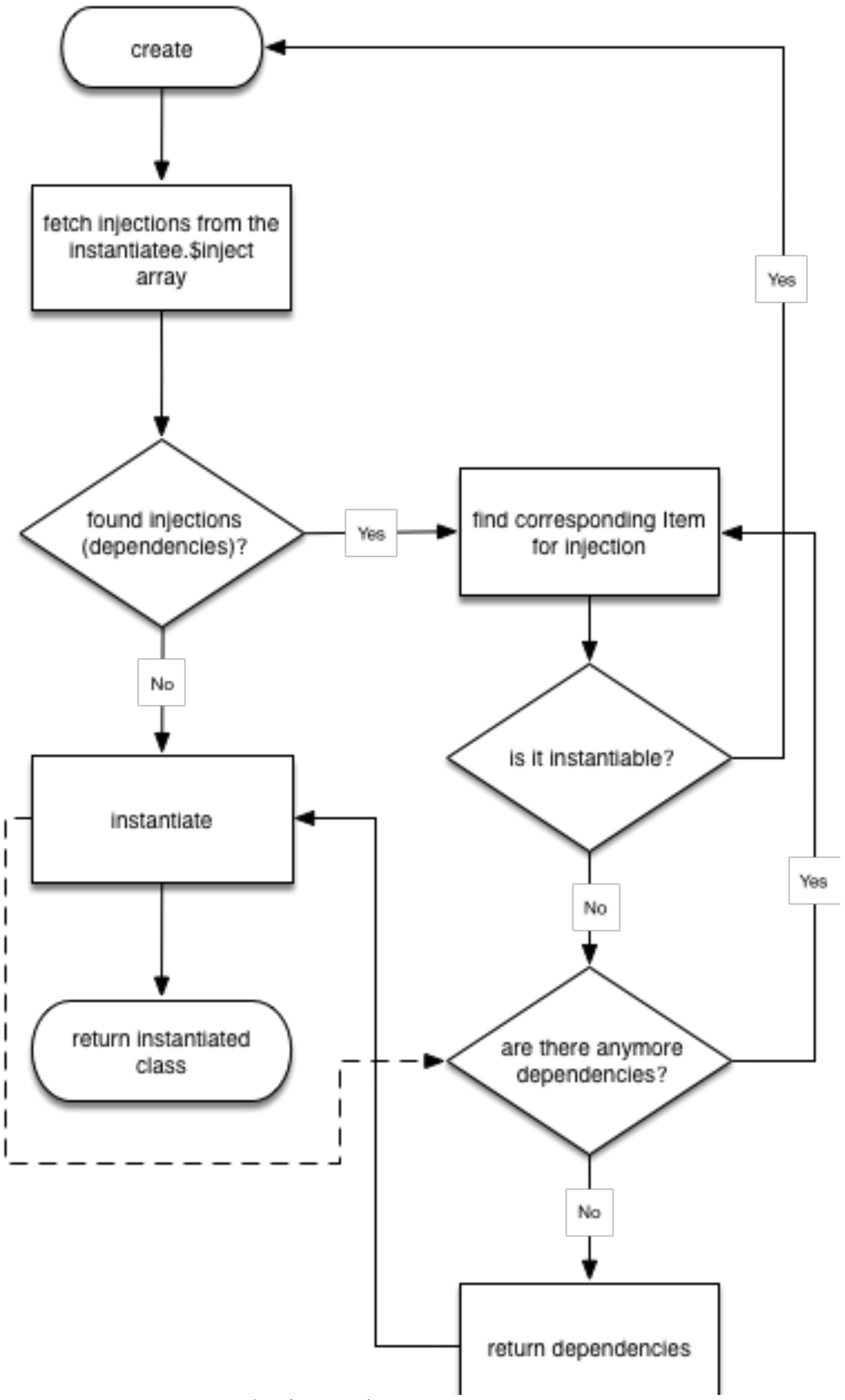

create
fetch injections from the instantiatee.$inject array
found injections (dependencies)?
Yes
No
find corresponding Item for injection
is it instantiable?
Yes
No
instantiate
return instantiated class
are there anymore dependencies?
Yes
No
return dependencies

Figure 11 Instantiating a new class (recursive)

Although not desirable neither JS nor TS provide the ability to statically detect the arguments that go in a class' constructor and hence the only way to determine the dependencies, is with a static array pertaining to the class, specifying the dependency names in order. (note that the dollar sign prefixing 'inject' was added to illustrate that this array is some kind of class metadata and not part of the class' main function).

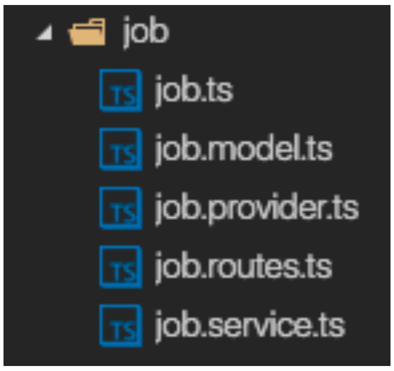
```
export class JobProvider implements IJobProvider {

    static $inject = [
                     "IJobService",
                     "ISensorService",
                     "IMessagingService",
                     "Logger"
                     ];

    constructor(private jobService: IJobService,
                private sensorService: ISensorService,
                private messagingService: IMessagingService,
                private logger: Logger) {
    }
```

Figure 12 The $inject array and constructor of the JobProvider class

3.5.5 System Control Flow

As previously mentioned, the project structure is split in a way that the code pertaining to one business function is grouped logically together, for the purpose of modularization. Within a module, there exists yet another structure that divides the execution of a request into various logical layers.

The overall pattern governing this system is the Model View Controller pattern or some kind of approximation to it. The view is of course omitted since this is a backend API that can serve various clients. The Controller is extended into various layers in order to keep the Single Responsibility Principle intact and making sure that one class is not doing more than it should. Below is the 'Job' directory, which will be used as an example to demonstrate this structure.



```
◢ job
    job.ts
    job.model.ts
    job.provider.ts
    job.routes.ts
    job.service.ts
```

Figure 13 The Jobs Directory

In Figure 13, apart from the obvious *job.model.ts*, the remaining files highlight the three different layers. The Router (controller), the Service and the Provider. On top is the router, this class handles the HTTP functionality pertaining to the management of Jobs. It will, decipher the request parameters or body into a JSON object, call the appropriate service or provider according to the request VERB and serve back the appropriate response.

The service layer is the one that does the business logic that pertains to the module in question. The `JobService` class, calls the underlying repository to store or retrieve jobs, and makes sure that they are returned in the correct order, etc.

It is very common, however, that a service would require another service to accomplish some tasks. When the list of service dependencies grows too large this tends to become unmanageable, especially when the service will start to depend on one another and thus creating circular dependencies. In order to alleviate this problem another layer was added, the provider layer. A provider acts as a façade (using the Façade Pattern), masking multiple function invocations on multiple service classes, that accomplishes one single atomic business function. Taking for example the function to create a new Job, the provider class `JobProvider` calls the service to create the Job in the database, then calls the `MessagingService` to register the Job with the messaging broker.

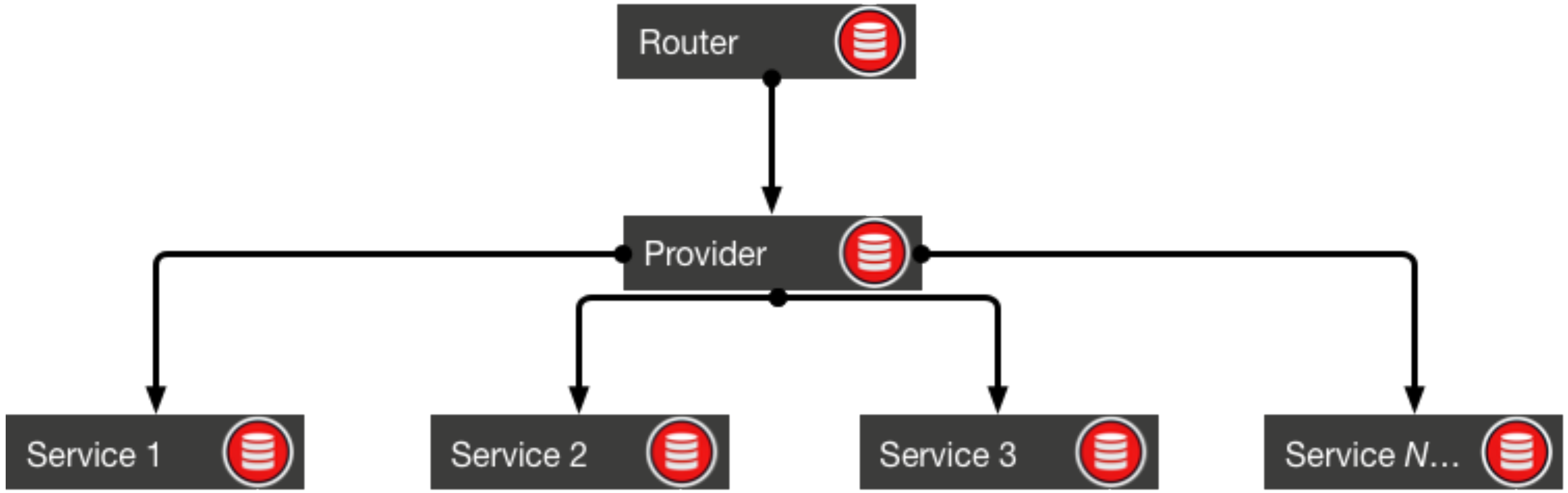


Figure 14 Facade Pattern

```
this.router.post("/job",
                async(req: any, res: Response, next:
NextFunction) => {
                    let request = req as
DependableRequest<JobRouterDependencyProvider>;

                    await request.dependencies.jobProvider
                    .validateJob(req.body.job)
                    .then((job: Job) => {
                        req.job = job;
                        next();
                    })
                    .catch((errs: Error[]) => {

res.status(HttpStatusCodes.BAD_REQUEST).json(errs);
                    });
                },
                async(req: any, res: Response) => {
                    let request = req as
DependableRequest<JobRouterDependencyProvider>;

                    await request.dependencies.jobProvider
                    .createJob(req.job, req.body.sensorId)
                    .then((job: Job) => {

res.status(HttpStatusCodes.OK).json(job);
                    })
                    .catch((err: Error) => {

res.status(HttpStatusCodes.INTERNAL_SERVER_ERROR).json(err.m
essage);
                    });
                });
}
```

Figure 15 The post route to create a Job

```
createJob(job: Job, sensorId: string): Promise<Job> {
    return this.jobService
    .createJob(job)
    .then((createdJob: Job) => {
        return this.sensorService
        .addJobToSensor(sensorId, createdJob._id)
        .then(() => {
            return createdJob;
        });
    })
    .then((createdJob: Job) => {
        this.registerJobWithBroker(sensorId, createdJob);
        return createdJob;
    })
    .catch((err: Error) => {
        this.logger.error(err);
        return Promise.reject(err);
    });
}
```

Figure 16 The createJob function in the JobProvider

```
createJob(job: Job): Promise<Job> {
    job.parameters = _.compact(job.parameters);

    return this.jobRepository
    .save(job)
    .catch((err: Error) => {
        this.logger.error(err);
        return Promise.reject(err);
    });
}
```

Figure 15 The createJob function in the JobService class

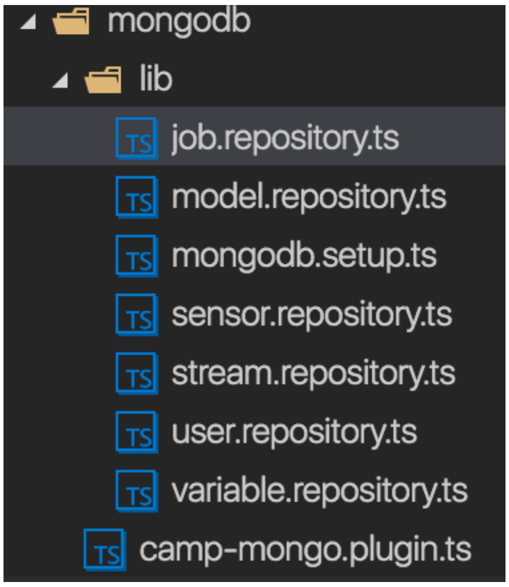
3.5.6 Plugins

In section 3.5.5 the execution of the business logic within the system was specified. In this section, the interaction with third parties will be explored. Three plugins have been implemented one plugin implements the interaction between the system and MongoDB in order to store and retrieve data from the MongoDB database. The second plugin is a plugin that implements messaging logic specifically to a RabbitMQ server via the AMQP protocol. The third plugin implements the messaging logic between the system and any MQTT broker.

Plugin: MongoDB

The database for this system was chosen to be MongoDB. This was chosen for multiple reasons. First, this system can be setup up to support multiple users, who can potentially have hundreds of devices, generating large amounts of data. In order to handle this potentially, the data might need to be spanning across different servers installed on different machines. Furthermore, although data integrity is important it is not as important as in the case of a banking system. Hence a NoSQL database was chosen to suit these needs (Chappell, 2015). This, however, can be easily remedied since if a separate operator using CampIoT wishes to use an SQL database, all that needs to be done is to implement an SQL plugin and installed in the system.

An Object Document Mapper, Mongoose, was used to alleviate repetition of database access and connection code. Furthermore, Mongoose allows the use to structure the code in Models and thus fitting into the MVC architectural pattern. A series of repositories were created, each corresponding to a module within the system.



```
mongodb
  lib
    job.repository.ts
    model.repository.ts
    mongodb.setup.ts
    sensor.repository.ts
    stream.repository.ts
    user.repository.ts
    variable.repository.ts
  camp-mongo.plugin.ts
```

Figure 17 MongoDb Repositories

Figure 17 illustrates all the repositories that are required by the system. In order to make it possible for the system to make use of the MongoDB plugin, without knowing about the implementation a set of interfaces had to be provided, that in turn had to be implemented by the plugin. These interfaces or contracts would eventually be exported into their own module that is installed by both the plugin and the system and this is how, through IoC the system would be able to use the plugins without writing any platform specific code.

```
export class JobRepository extends MongooseModelService<Job>
implements IJobRepository
```

Figure 18 The signature of the JobRepository

Each repository defines its Model Schema and registers it to Mongoose. Mongoose then simply translates the JavaScript Objects supplied to a document in MongoDB following the schema provided.

```
private static modelDefinition = mongoose.model("Job", new
Schema({
    name: String,
    parameters: [Schema.Types.String]
}));

constructor() {
    super(JobRepository.modelDefinition);
}
```

Figure 19 Job Schema Specification

Since most of the database interaction logic is being handled by mongoose the most common functionality can be extracted into a single abstract class. Hence, the class `MongooseModelService` was created that has methods to save data in a model and retrieve data pertaining to a model type.

```
export abstract class MongooseModelService<T extends
CampModel> {

    constructor(protected modelDefinition: Model<any>) {
        mongoose.Promise = <any>global.Promise;
    }

    protected saveModel(model: T): Promise<Document> {
        let generatedModel = new this.modelDefinition(model);

        return (<any>generatedModel).save()
        .catch((err: WriteError) => {
            let error: StorageError = {
            code: err.code,
            message: err.errmsg || (<any>err).message,
            name: "Save"
            };

            return Promise.reject(error);
        });
    }

    protected findManyModel(queryObject: any):
    Promise<Document[]> {
           return
           <any>this.modelDefinition.find(queryObject).exec();
    }

        protected findOne(queryObject: any): Promise<Document>
    {
           return
           <any>this.modelDefinition.findOne(queryObject).exec
           ();
    }
```

Figure 20 MongooseModelService Snippet

Plugin: AMQP (RabbitMQ)

Just like for storage plugins an interface was developed for Messaging. This interface is the *IBrokerService* interface. It contains six methods that must be implemented:

- `registerSensor`
- `registerJob`
- `publish`
- `subscribe`
- `unsubscribe`
- `cleanUp`

Implementing these six methods should cover all the messaging requirements. The AMQP plugin implements these by using a library called *rabbot*. *Rabbot* is an NPM package that handles communication via AMQP to RabbitMQ.

Apart from simply interacting with the broker the messaging plugin must also determine how to connect, what architecture to use and best practices. Following the AMQP features, it was decided that a sensor would have its own exchange. Whenever a new sensor is registered a new exchange will be created using the *sensorId* as the exchange name. This was done to be as scalable as possible. Different sensors may be able to reside on separate servers and have whole servers dedicated to them. Variables and Jobs will be registered as queues and topics.

A sensor wishing to publish some information about a variable it has registered with the system, should publish on the topic named with its ID. The system will be subscribed on the same topic name and will handle messages from that variable accordingly. The same process in reverse is done for triggering Jobs on the sensor.

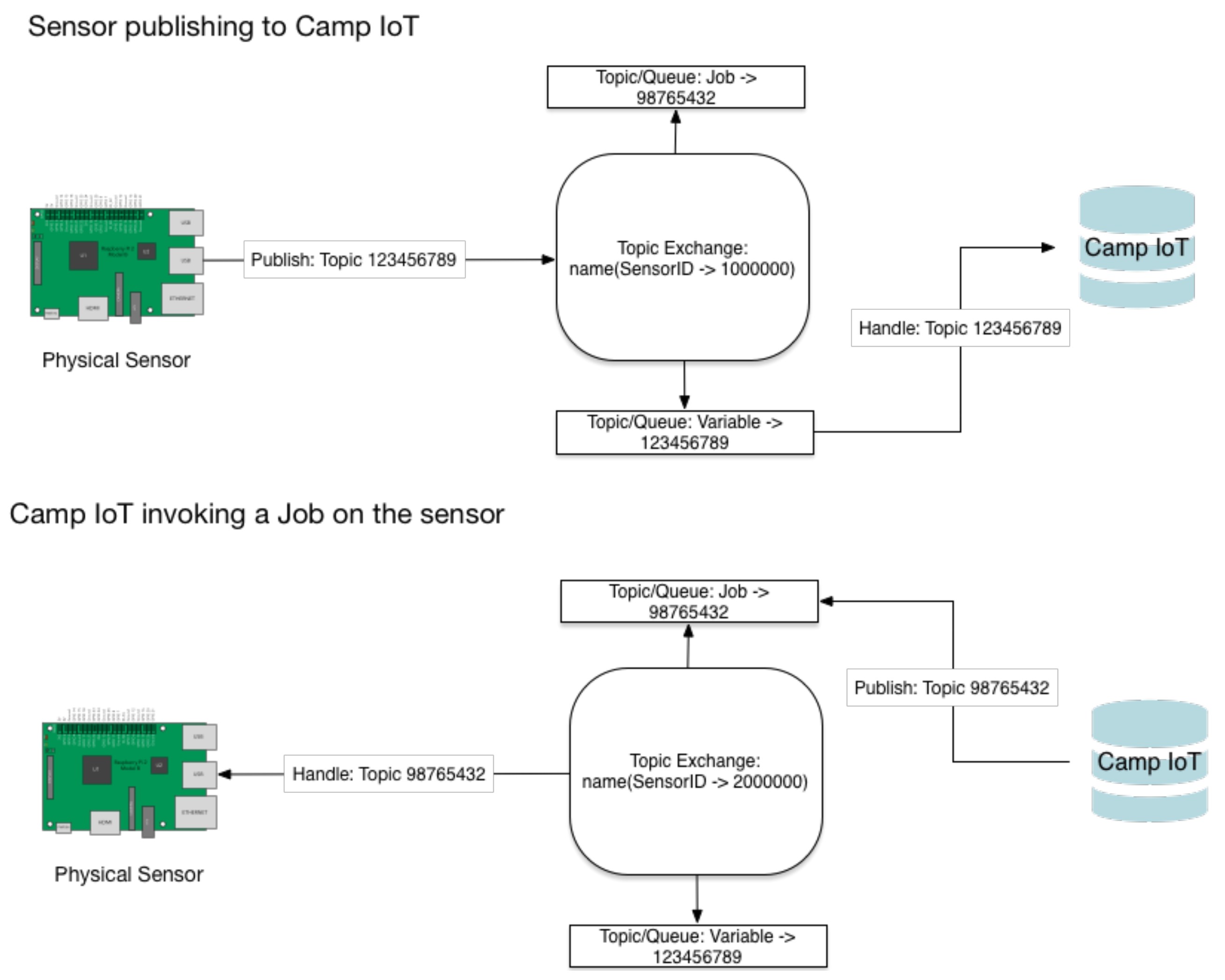

Sensor publishing to Camp IoT
Topic/Queue: Job -> 98765432
Publish: Topic 123456789
Topic Exchange: name(SensorID -> 1000000)
Camp IoT
Handle: Topic 123456789
Physical Sensor
Topic/Queue: Variable -> 123456789
Camp IoT invoking a Job on the sensor
Topic/Queue: Job -> 98765432
Publish: Topic 98765432
Handle: Topic 98765432
Topic Exchange: name(SensorID -> 2000000)
Camp IoT
Physical Sensor
Topic/Queue: Variable -> 123456789

Figure 21 AMQP System Architecture

Plugin: MQTT

The MQTT plugin works in very much the same way as the AMQP plugin works. The only difference is that MQTT is a much simpler protocol, it has no concept of queues and exchanges. Hence, the code is much simpler, publish to a topic and subscribe to a topic. The same principles as that of the AMQP implementation applies i.e. the topic name is the variable or job ID.

3.5.7 Bootstrapping the Application

The key to understanding how the application works is to understand how the application is being bootstrapped. After the Gulp Build is done (see 3.5.3) the project directory becomes the '_build' directory. In here, the compiled JS files can be found. The '*App.js*' whose equivalent is the '*App.ts*' file (in TypeScript) is the starting point of the program. This file is responsible of calling the 'app-setup' directory/namespace. As the name suggests this namespace contains setup functions that configure different parts of the system.

App-Setup: Container-setup

The container-setup file contains code to create and populate various IoC containers (see **Error! Bookmark not defined.**). There are three functions in this file each one creating a separate container.

It is good to note that registering a *ContainerItem* can be done via a fluent interface (Fowler, 2005). This has been done to simplify and make the declaration more readable. Below is an example of such declaration.

```
ContainerItem.create(container)
.forKey("IAuthService")
.use(AuthService)
.asSingleton()
.andRegister();
```

Figure 22 Create a ContainerItem, set it up and Register it

The first function `setupSystemContainer` creates a container with minimum amount of *ContainerItems* mainly those needed to get the app's configuration and the logger so that it can log some initial tracing logs.

The function `setupOperationalContainer` sets-up the container that will be used during the execution of a request, while the function `setupBrokerContainer` sets up a container that will be used within the messaging process. More details in section *App-Setup: Broker-Setup*. This is needed because the messaging process lives in a separate process entirely and hence a different container is needed since memory can't be shared between processes. Furthermore, not all items are needed in the messaging process, and thus setting up a different container with less items reduces the memory footprint.

App-Setup: Database-Setup

As the name of this file implies the function inside the 'database-setup' file initializes the connection to the database. Since the controller cannot depend on a concrete implementation of the storage facilities, yet another storage contract has been created in order to call `initConnection.` This has to be implemented by the plugin who should know how to open a connection to the database in use.

App-Setup: Broker-Setup

The *broker-setup* file is perhaps the most peculiar of all the other setup files. This is because unlike other traditional languages like C# and Java, JavaScript and therefore NodeJS is single threaded, where an event based loop is executed giving the illusion of asynchronousness. There are many pros and cons to this approach. Since the system is meant to run in a cloud environment (although it can run on a traditional setup just as well) it is useful to allow the system to take advantage of the setup and move towards distributed computing. There are different ways to achieve this and although it is not the goal of this project, a separate process is forked to handle all communication between Camp and Sensors.

As soon as the default function in this file is executed, a new process is forked using the *WorkerService* which is a class that is implemented to wrap functionality to fork processes and handle communication between processes. The messaging process starts a socket IO connection, which would be used for real-time updates to the dashboards. It also starts listening to inter-process messages upon which (depending on the command) it will start handling messages or publish messages using the *BrokerService.*

App-Setup: Diagnostics-Setup

The default function that executes when the *'diganostic.setup'* file is invoked, is a very simple function that sets-up logger, to provide logs in console when in development mode and to a file when in production mode.

App-Setup: Routing

Express.js implements a very easy to use Router that allows the developer to create routes using the correct HTTP VERBS very easily. In order to adhere to the Single Responsibility Principle, a Router class has been implemented for every module of the system. A Router class creates an Express router and registers the routes pertaining to a particular module while providing the handlers for the routes.

A Router class must extend the `RouteController` class. This is a very simple abstract class that registers an Express middle ware in the constructor. This middleware has a very important role in the IoC/Plugin mechanism. This is because this middleware runs as soon as a request comes in and before all the other middleware/routes are executed. This middleware sets up a new container per request and assigns it to the request object, which is later on passed on to the rest of the route handlers and middleware. This means that dependencies are unique (transient) per request. This keeps the truly, desirable, stateless nature of HTTP i.e. no objects are being kept and session management is not needed, reducing this overhead and allowing the middleware to be able to scale and be distributed easily across machines.

```
export abstract class RouteController<T> {
    protected router: Router;

    constructor(private dependencyProvider: Function) {
        this.router = express.Router();

        this.router.use(this.routePrefix(), async (req: 
Request, response: Response, next: NextFunction) => {
            let container = setupOperationalContainer();
            let request = req as DependableRequest<T>;
            request.dependencies = 
container.create<T>(<any>this.dependencyProvider);

const contextService = 
container.getByKey<IContextService>("IContextService");
            contextService.serverAddress = 
req.header("host");

            next();
        });
    }

    protected abstract routePrefix(): string;

    get routes(): Router {
        return this.router;
    }
}

export interface DependableRequest<T> extends Request {
    dependencies: T;
}
```

Figure 23 The RouteController class

```
export class ControllerInfoRouter extends
RouteController<ControllerInfoRouterDependencyProvider> {

    static $inject = ["IControllerInfoService"];

    constructor() {
        super(ControllerInfoRouterDependencyProvider);

        this.createGetRoutes();
    }

    protected routePrefix(): string {
        return "/controller-info";
    }

    private createGetRoutes(): void {
        this.router.get("/controller-info", async(req: Request,
res: Response) => {
            let request = req as
DependableRequest<ControllerInfoRouterDependencyProvider>;

            await
request.dependencies.controllerInfoService.getInfo()
            .then((controllerInfo: ControllerInfo) => {

res.status(HttpStatusCodes.OK).json(controllerInfo);
            })
            .catch((err: Error) => {

res.status(HttpStatusCodes.INTERNAL_SERVER_ERROR).json(err);
            });
        });
    }
}

class ControllerInfoRouterDependencyProvider {
    static $inject = ["IControllerInfoService"];

    constructor(public controllerInfoService:
IControllerInfoService) {
    }
}
```

Figure 24 The ControllerInfoRouter extending the RouteController

3.5.8 Messaging

The messaging algorithm is the most important and the most complex feature in the system. As stated earlier on in this text the `IBrokerService` interface specifies all the required methods to register sensors, register Jobs and Variables, subscribe to messages and publish messages. However, this is the logic to interact with the broker. There exists logic to make sure messages get to the `IBrokerService`.

Before going through the messaging flow, there are two services[6] that need to be described in some detail. The `WorkerService`, already mentioned in section 3.5.7 provides functionality to fork processes and keeps a reference to them in order to send and retrieve messages between the parent process and the sub processes. The `MessagingService` is the service that is actually used by providers to invoke the `BrokerService`. The messaging service uses the `WorkerService` to send messages or commands that will eventually be interpreted by the messaging process, which will in turn call the `BrokerService` accordingly. The messaging service sends a worker message, which is comprised of a Command *enum*, denoting whether this message should register as sensor, variable or job, whether it should publish a message or whether it should subscribe to a topic. The worker message contains a payload that consists of various attributes, namely a data attribute and a task key.

```
export interface WorkerMessage {
    command: Command;
    payload: WorkerMessageData;
}

export interface WorkerMessageData {
    topic?: string;
    handlerId?: string;
    sensor?: string;
    data?: any;
    taskKey?: string;
    isRemovableHandler?: boolean;
}
```

Figure 25 The Worker Message with Payload Object

[6] 'Service' means the service layer in the Repository Pattern that this system implements.

Upon calling the `createSensor` function on the *SensorProvider* the provider will call the messaging service's function `registerSensor.` This will call the worker service, which will send a message to the messaging process, which will in turn call the broker service's `registerJob`. The same thing happens when a job is created.

Creating a variable, however, is a bit more complex. When a variable is created, a subscription would need to be initiated with the broker. However, a subscription needs to know how to handle a message as it comes along. It might be the case that handling a message is quite complex because there is some business logic to be performed, depending on which message is being handled. It is also possible to have different handlers that handle the same message differently. Furthermore, to adhere to the SRP, this logic should not be crammed in the messaging process; it should be contained within the appropriate module example: if there is handler logic pertaining the Variable module it should be under the variable module.

A `TaskService` was created in order to register tasks, which handlers can execute and to retrieve them. The task service is none other than a *HashMap* wrapper. This task service is injected through IoC to classes denoted with '*[module-name]-task.provider*'. Task Providers are recipients of message handlers pertaining to one module. In this case, there is only one Task Provider class specified and this is the *variable-task.provider* or *VariableTaskProvider*. This is because only variable messages need to be handled as of yet.

The *VariableTaskProvider* contains two tasks, `handleVariableUpdate` and `handleVariableSiteUpdate.` The former is used to store variable status updates in the database, whereas the latter is used to send status updates to the dashboard via websockets. Both of these tasks handle the same message in different ways.

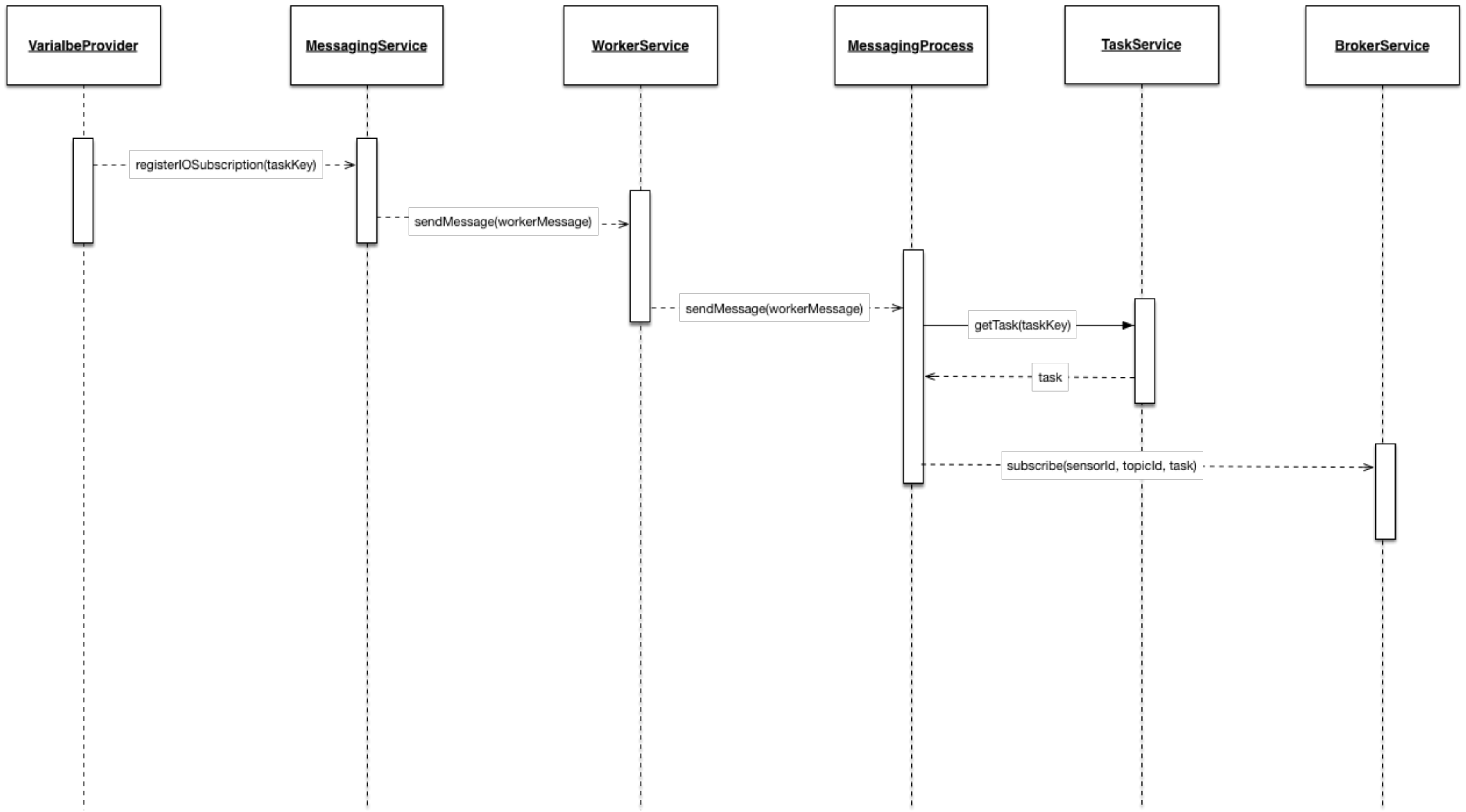

VarialbeProvider
MessagingService
WorkerService
MessagingProcess
TaskService
BrokerService
registerIOSubscription(taskKey)
sendMessage(workerMessage)
sendMessage(workerMessage)
getTask(taskKey)
task
subscribe(sensorId, topicId, task)

Figure 26 Registering a new Variable and subscribing the handler

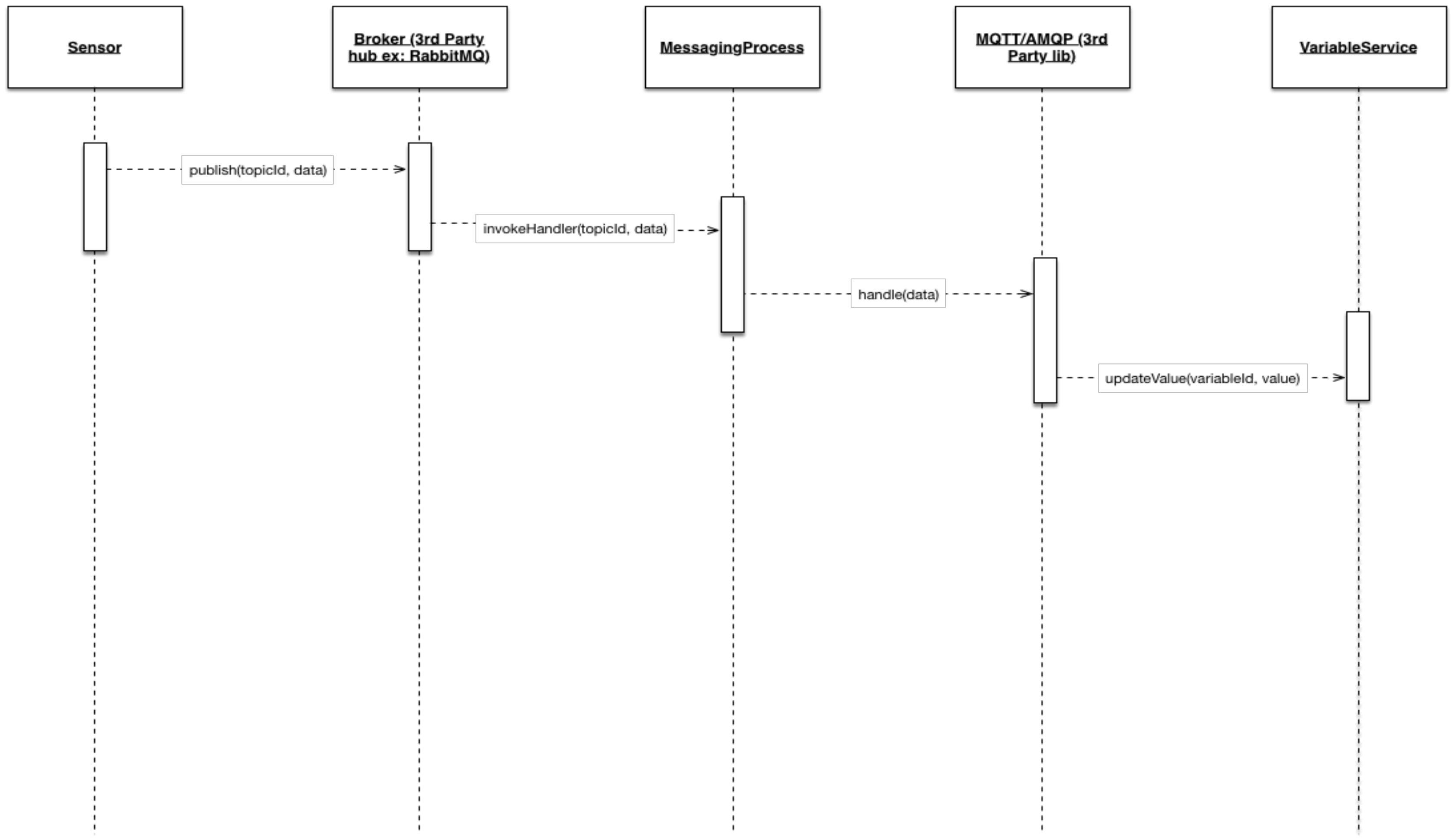

Sensor
Broker (3rd Party hub ex: RabbitMQ)
MessagingProcess
MQTT/AMQP (3rd Party lib)
VariableService
publish(topicId, data)
invokeHandler(topicId, data)
handle(data)
updateValue(variableId, value)

Figure 27 Handling a Message

Figures 24 and 25 illustrate via a Sequence Diagram the process of registering a Variable and handling of a variable update coming from the sensor. The variable registration was selected because it is the most complex. Naturally registering the sensor itself and registering jobs is very similar omitting the *TaskService* process, since neither of these processes requires handlers.

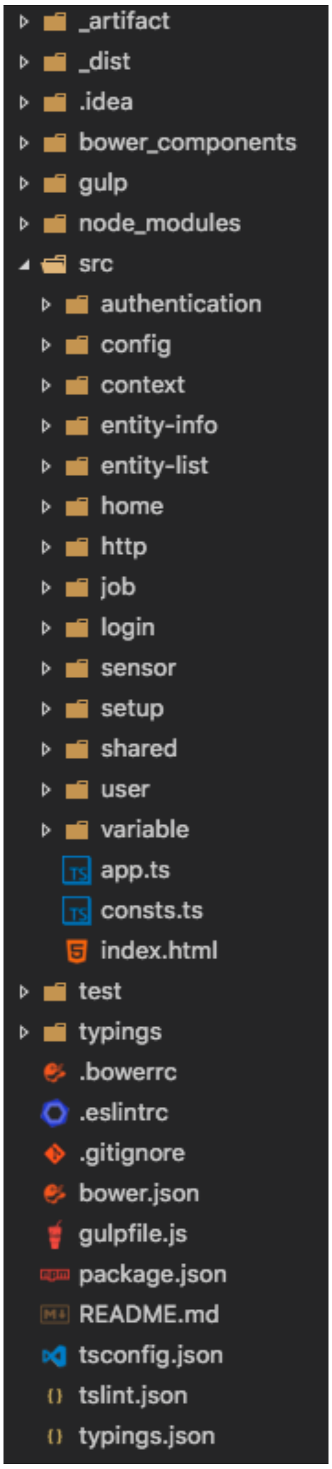
3.5.9 The Dashboard

Although the system is a web standards compliant REST API, which means it can accept virtually any client that can communicate over the HTTP protocol, this project implements a web app client in order to demonstrate the capabilities of this system.

In the frontend world more specifically the JavaScript ecosystem, nowadays there exist ample amount of frameworks and tools, each trying to solve either a different problem or the same problem with a different approach. Furthermore, these frameworks usually become outdated as more programming patterns are incorporated into newer frameworks. Hence, thorough research had to be done to choose the right framework. In the end, AngularJS was chosen as the backbone framework of the project. Although these are JavaScript frameworks, just like the backend counter-part of this system, the JavaScript language was yet again augmented by Typescript in order to detect errors while the program is being written as opposed to detecting them at runtime.

Since this app is targeted towards controlling Smart Environments remotely, it is very desirable that the app is targeted towards mobile devices. This requires a responsive app layout that can resize and nicely fit in different screen sizes and shapes. For this reason, a styling framework was required to alleviate the need to create common components like buttons, forms, etc. and the need to create custom styling code to handle responsiveness. Foundation for Apps is a framework that provides this functionality and it is also built around the AngularJS ecosystem. Furthermore, one of the three sub frameworks of Foundation the "for Apps" framework, is as the name implies targeted for mobile and phone apps.

With these two frameworks as the backbone of the project several screens where devised, in order to show the functionality of the Camp IoT Controller. The project structure is very similar to the backend structure. It groups the different components or modules of the system in together, and thus making each module self-contained.



```
▸ _artifact
▸ _dist
▸ .idea
▸ bower_components
▸ gulp
▸ node_modules
▾ src
    ▸ authentication
    ▸ config
    ▸ context
    ▸ entity-info
    ▸ entity-list
    ▸ home
    ▸ http
    ▸ job
    ▸ login
    ▸ sensor
    ▸ setup
    ▸ shared
    ▸ user
    ▸ variable
      app.ts
      consts.ts
      index.html
▸ test
▸ typings
  .bowerrc
  .eslintrc
  .gitignore
  bower.json
  gulpfile.js
  package.json
  README.md
  tsconfig.json
  tslint.json
  typings.json
```

Figure 28 Dashboard Project File Structure

"Gulp" was also used to transpile the TS files and place the JS files in the *_dist* folder. However, the dashboard requires some more complexity in its build process and this is because assets/content/images and html template files need to be copied to the *_dist* folder. Furthermore, SCSS files need to be transpiled to CSS files and placed in the *_dist* folder.

Just like the Controller, the file that bootstraps the application is the *app.js.* However, since this is a web application the file that is served first by the webserver is the *index.html* file, which in turn contains a reference to the *app.js* script.

The app uses UI Router. This AngularJs module allows routing and thus navigation within an AngularJs single page application. This router also changes the url and hence allows a user to navigate directly to a screen using the url just like normal web pages which this is harder to achieve in a SPA.

Dashboard: Flow

The Dashboard has three main screens apart from the login screen and the Home Screen. The home screen is a very simple screen displaying information about the backend controller for the purposes of this project. This is used to identify the different operating systems and communication protocol that are used in order to demonstrate that the system is really hosted on different platforms. There exists a flow of how the user should create register sensors, variable and jobs. This is illustrated below.

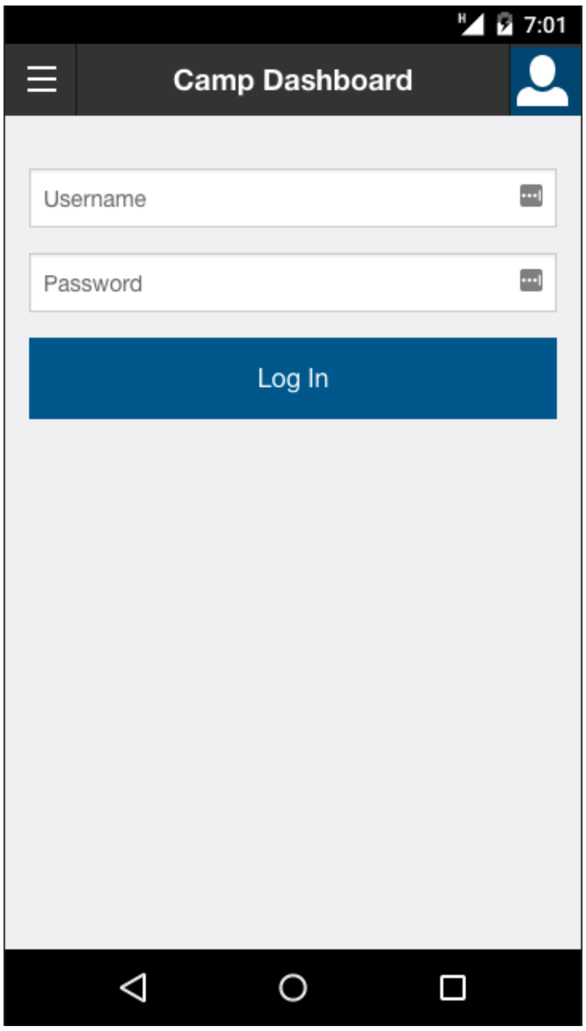


Figure 29 The Login screen

The login screen is where the users starts their journey in the app. It is a simple login screen that accepts an email address and a password and proceeds to authenticate itself with the controller. The controller returns a JSON Web Token that is stored in localstorage and sent as a header in all subsequent requests to the controller.

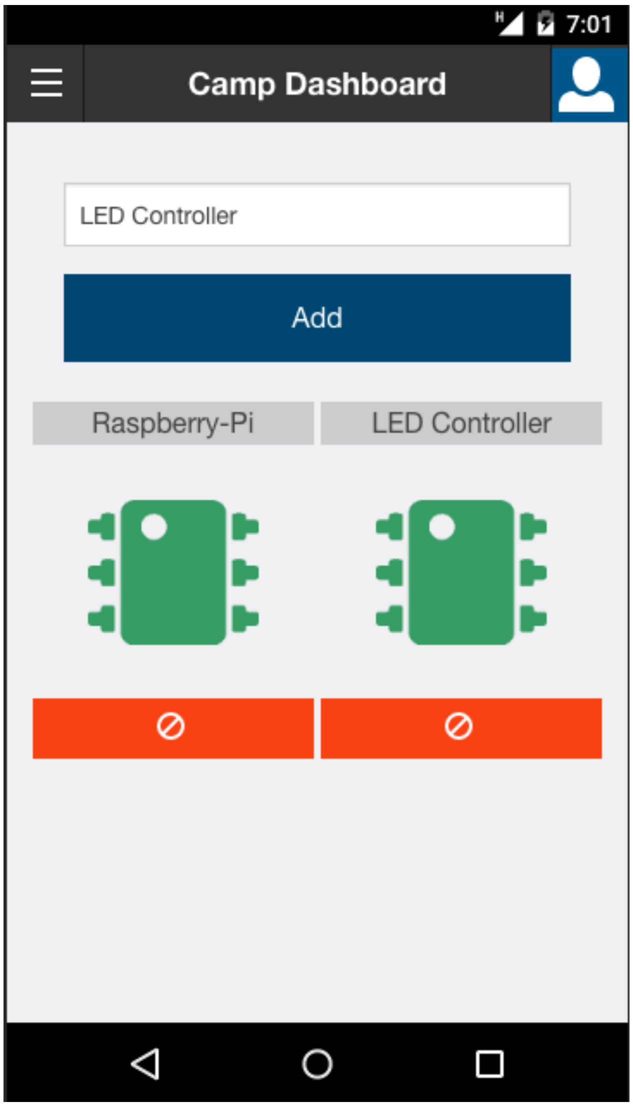


Figure 30 The Sensor Screen

The sensor screen is a simple screen that accepts a name, which will create a new sensor in the system. The Sensor screen is comprised of a text box, a submit button and an *entity-list* component. The latter is custom built HTML5 components that accepts an array of entities (in this case sensors) and displays them as a set of responsive boxes in a grid, with each box being clickable and able to route you to the Sensor Detail screen. Each box contains a button that is able to delete a sensor from the system.

```
<entity-list entities="vm.sensors"
    entity-detail-state="'secure.sensor-detail'"
    entity-controller="vm"
    image-src="'images/chip.png'">
</entity-list>
```

Figure 31 Using the entity-list component as a standard HTML 5 web component

Clicking a sensor takes you to the Sensor Detail screen. This screen displays the id of the sensor together with a button, which copies it to the clipboard. This id is there in order to be used in the sensor to publish data. Two buttons are available to allow the user to add a variable or a job respectively. Below are two entity-list components that display the variable and jobs respectively belonging to the sensor. This is demonstrating the reuse of the *entity-list* component.

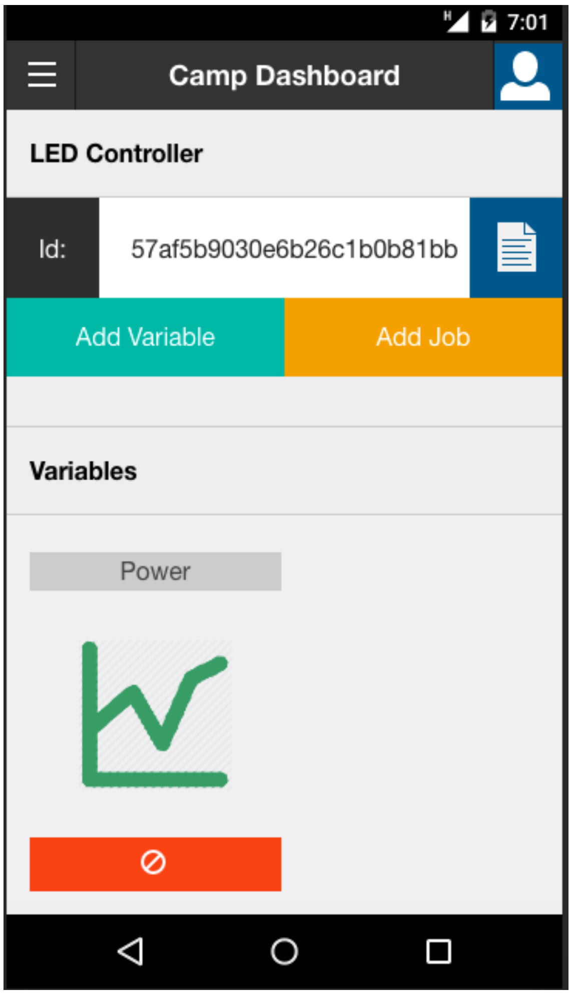


Figure 32 The sensor detail screen showing one variable

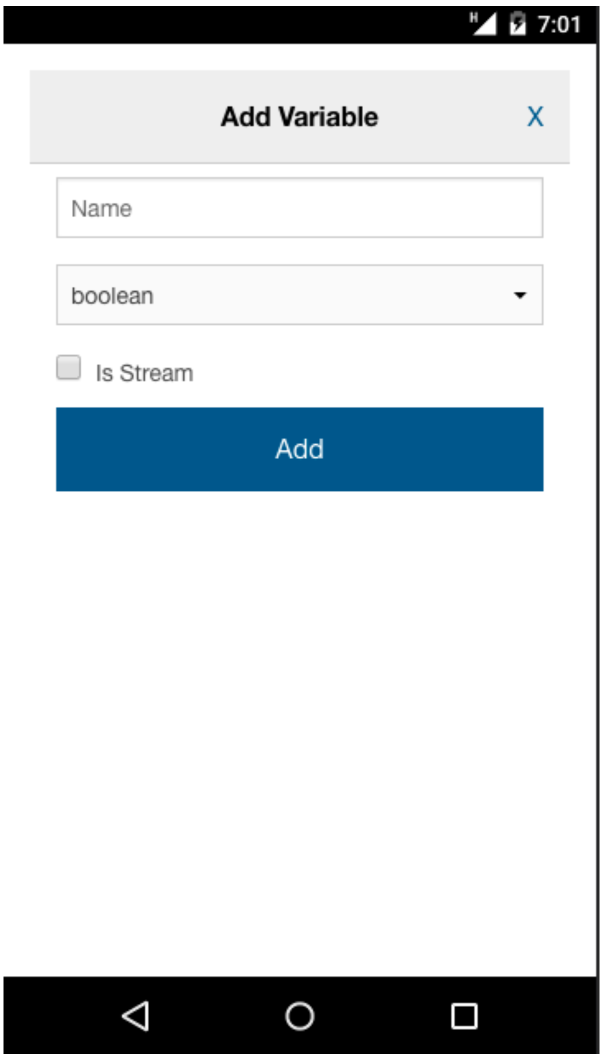


Figure 33 Add Variable Dialog

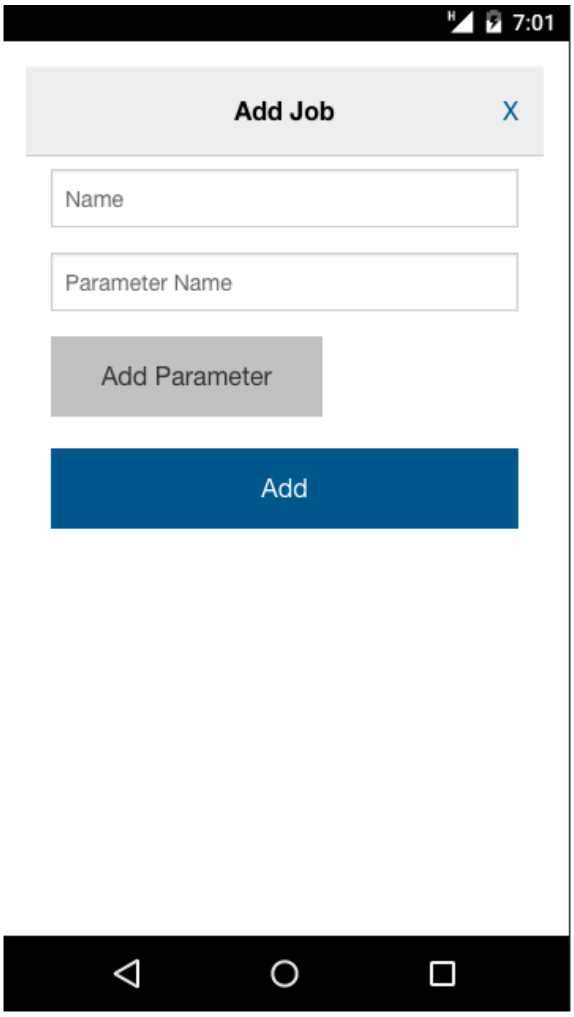


Figure 34 Add Job dialog

The Variable and Job Detail screens are very similar to the Sensor detail screen. They each contain the Id container and the copy button that copies the id to the device's clipboard. Hence, an *entity-info* containing these two components custom component was created in order to be shared with all the screens.

```
<entity-info entity="vm.variable"></entity-info>
```

Figure 35 Using the entity-info component

The variable detail screen is made up of three components, the variable information, the variable current state component and (if it is a streaming variable) a graph showing past history of the variable states (up to the last twenty for browser rendering performance reasons and UI limitations).

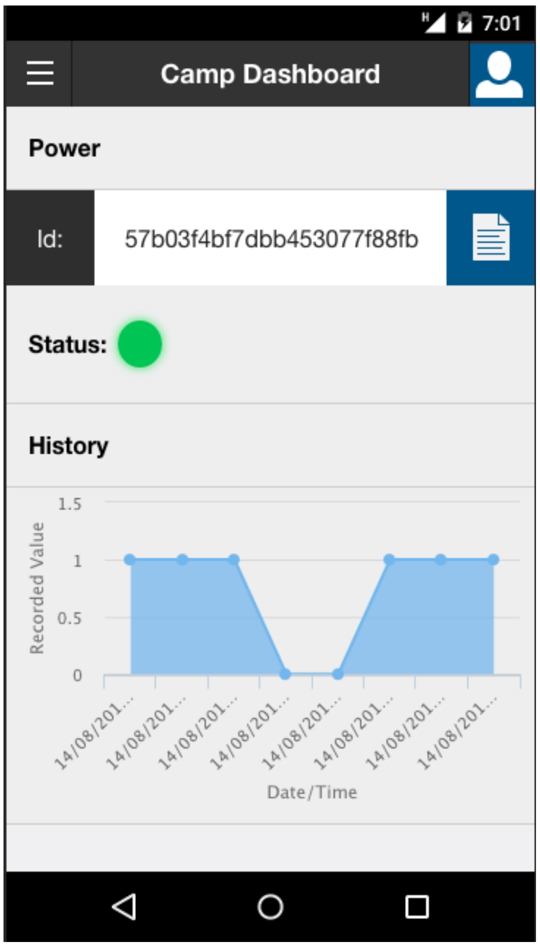


Figure 36 The variable detail screen of a streaming variable

This screen is also capable of displaying real time updates to the current state of the variable. It does this by using SocketIO. The same library that is used in the backend controller, this time is used from a client perspective. On load, the screen calls the API to supply it with the socket server information, which is in turn supplied to the SocketIO library to open a web socket to receive real time updates section 3.5.8 illustrates how a handler is created at this point to handle messages coming from the sensor. At this point two handlers are registered with the broker (for which logic is found in the *VariableTaskProvider*), one handling saving the data to the database and the other pushing the data through SocketIO.

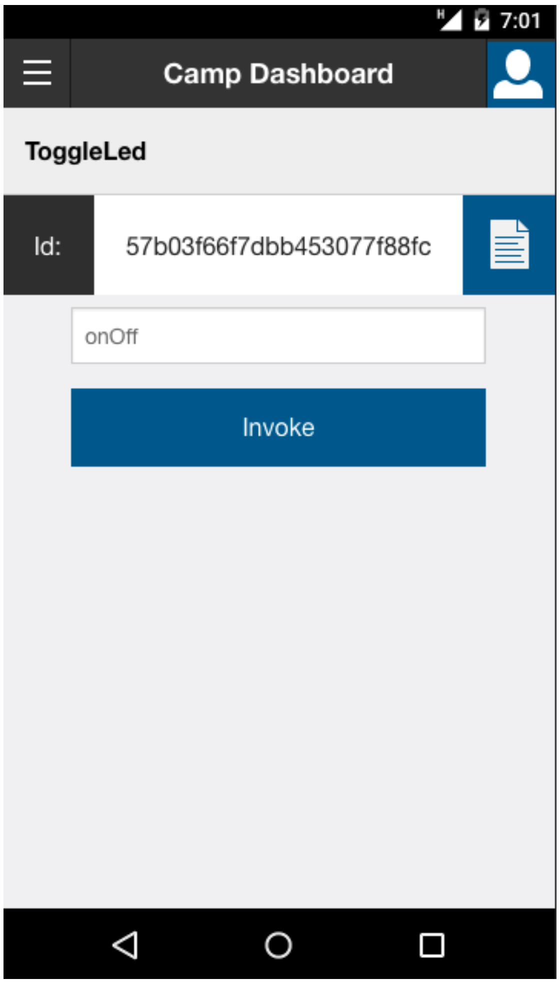
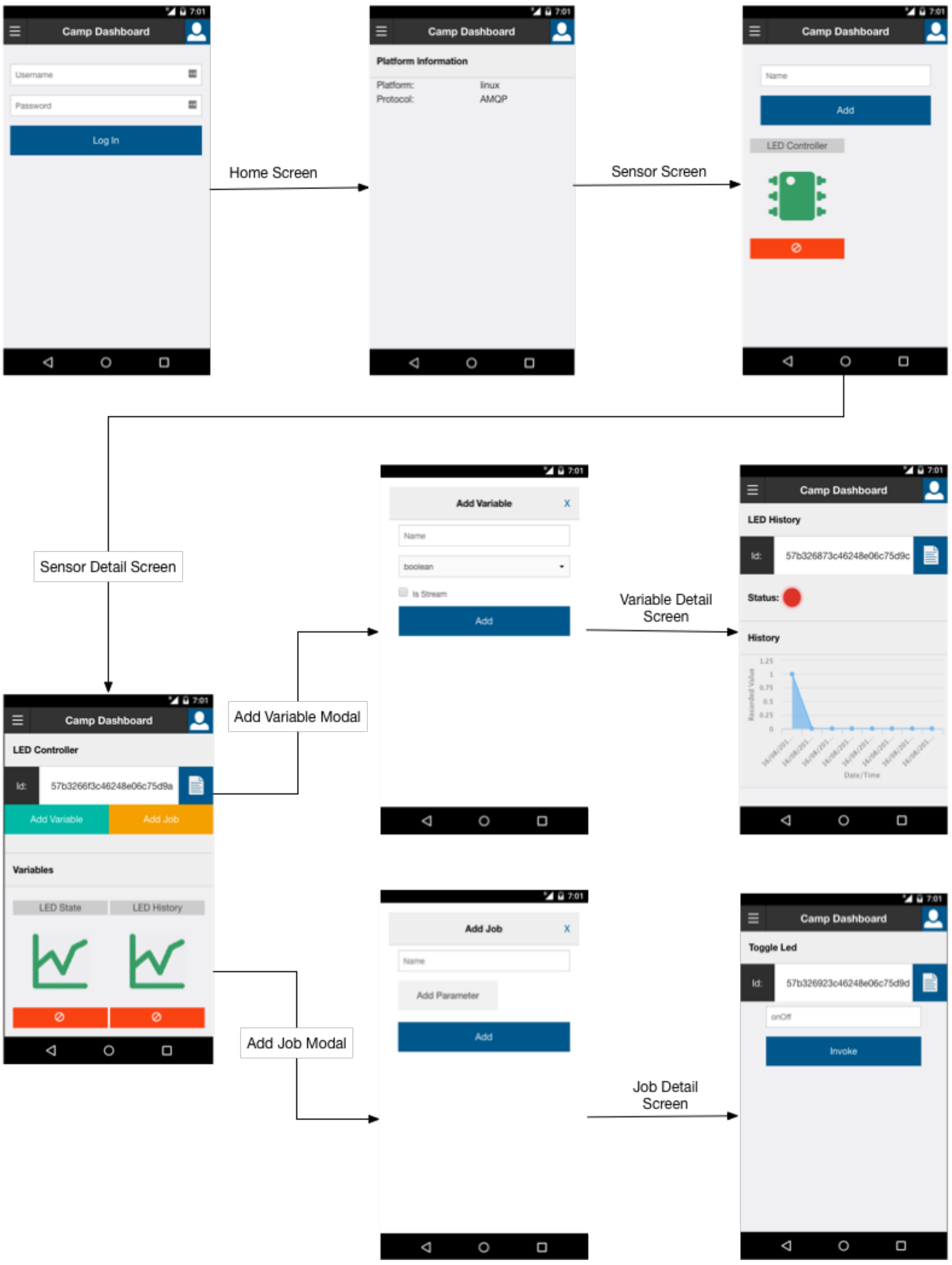


Figure 37 Job Invocation screen

The Job Detail screen or Job Invocation screen is a very simple screen making use of the *entity-info* component to display the id and the copy to clipboard button. It also creates a list of text boxes per parameter that a job requires. The invoke button build a hashmap containing the data parameter name as key and the actual value input by the user as value. At this point the controller would generate a publish message to the broker supplying the parameter data as the body of the message which in turn is handled by the sensor accordingly.

Dashboard: Flow Diagram



Dashboard: Packaging

The advent of smart phones coupled with the internet has been prime factors in the move to the cloud of most software services. It is for this reason that a mobile app is deemed as the most fitting of mediums of the dashboard. However, the dashboard is not the prime objective of this project and hence, it was deemed more appropriate to focus on other aspects of the system. This led to the dashboard to be a web based application, which is not the most sought after medium on mobile phones but it is accessible from virtually any modern device (phones, desktops, TVs, etc.) thus making it the best candidate. However, to make it more easily accessible from a smart phone, Cordova was used to wrap the web app into a native app that can be installed on the phone (in the form of an APK).

Chapter 4: Conclusions and Future Work

4.1 Experiment Setup

To demonstrate the effectiveness of this project an experiment has been setup to determine whether the system managed to achieve the goals it set out to tackle i.e. it is deployable on different platforms and different cloud systems (including database and communication protocol).

Figure 36 illustrates the experiment setup; this shows quite a complicated setup with various parts of the system working on different servers on different cloud providers. This overly complicated setup was done to demonstrate that the system makes full use of cloud services, taking advantage of different systems and thus ensuring that it makes use of the best aspects of different providers while also attaining the lowest setup costs possible.

The first setup has the controller setup on an Amazon Linux EC2 virtual machine, a Dashboard (titled *Dashboard 1*) setup on an Azure Web App Application, a RabbitMQ broker setup up on CloudAMQP (hosted on Azure), Mongo Lab MongoDB instance setup on Azure and a Raspberry-Pi setup locally connecting to the RabbitMQ broker.

The second setup has the controller setup on an Azure Windows Server VM, a Dashboard (title *Dashboard 2*) setup on an Azure Web App Application, a MQTT broker setup on CloudMQTT (hosted on Heroku), Mongo Lab MongoDB instance setup on Azure and a Raspberry-Pi setup locally connecting to the MQTT broker.

There was no need to demonstrate that the dashboard could be deployed on different environments since the web app is simply made up of a collection of HTML, JS and CSS files, all of them standard web files that any web server can serve. Hence, taking into consideration the familiarity of the Azure framework, both instances of the Dashboards where deployed to Azure to alleviate the burden of having to setup multiple machines with different features and capabilities. Furthermore, it demonstrates that the controller and dashboard do not need to be hosted on the same environment at all and are totally independent of each other.

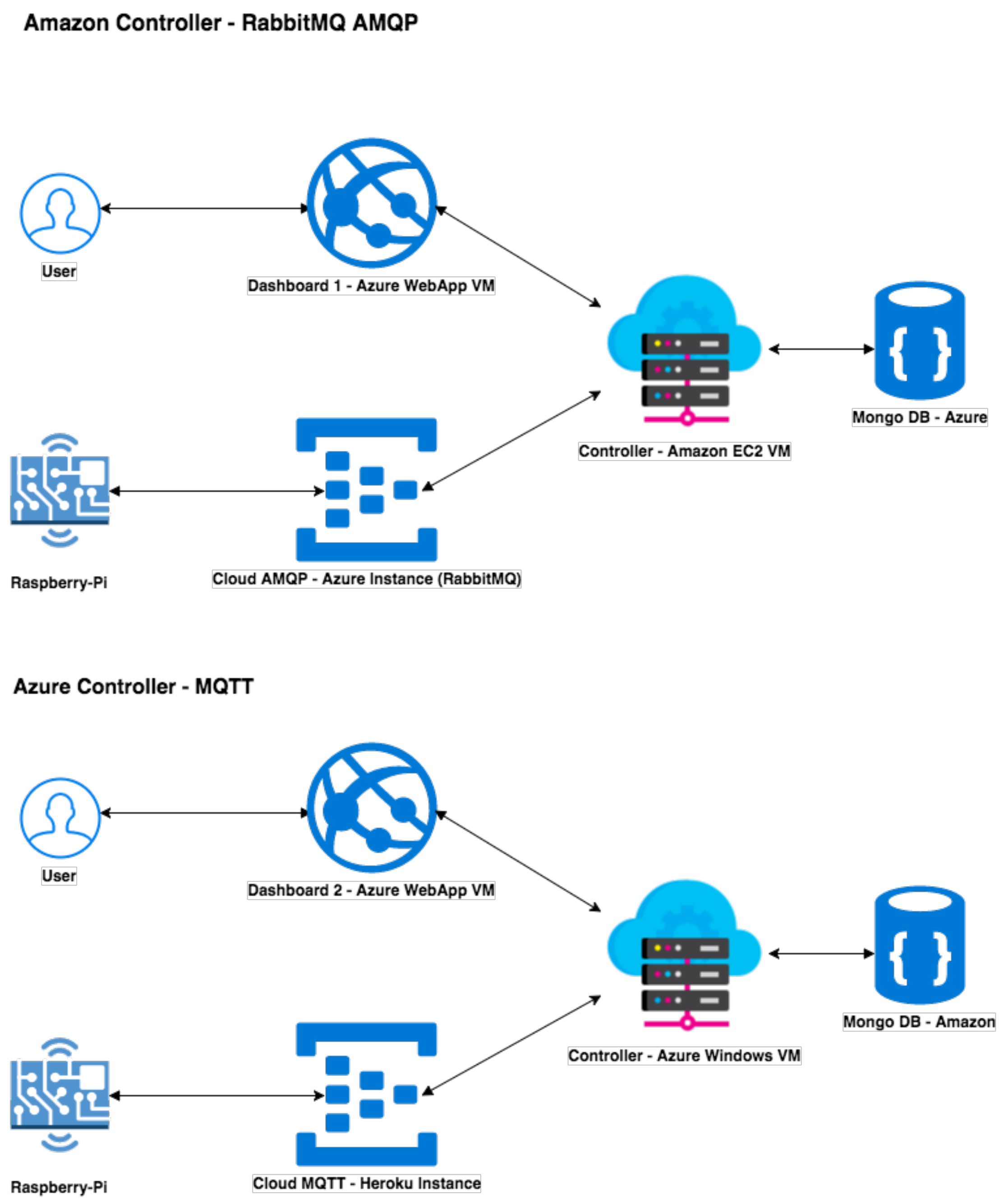

Amazon Controller - RabbitMQ AMQP
User
Dashboard 1 - Azure WebApp VM
Controller - Amazon EC2 VM
Mongo DB - Azure
Raspberry-Pi
Cloud AMQP - Azure Instance (RabbitMQ)
Azure Controller - MQTT
User
Dashboard 2 - Azure WebApp VM
Controller - Azure Windows VM
Mongo DB - Amazon
Raspberry-Pi
Cloud MQTT - Heroku Instance

Figure 38 Experiment Setup

4.2 Results

The experiment documented here: https://youtu.be/XIZnHg0f5gQ demonstrates that the system works, that jobs can be invoked and variable state can be viewed from different platforms and communication protocols.

The video shows how a sensor was registered to the different setups together with a job that can be invoked, which triggers the LED on or off state, depending on the parameter value input by the user. The sensor in each setup, sends a state updating the variable and thus the informing the user in real time. A stream of the historical value of the LED state can also be read in real time. (The sensor updates its state every 10 seconds).

4.3 Evaluation and Conclusion

This project has successfully demonstrated that the problems that the project set out to resolve have been achieved. All the aims were covered, the Cloud Provider and platform lock-in where resolved by introducing a REST based web API middleware that abstracts away the underlying platform together with a plugin based architecture that allows the core logic of the system to remain unaltered when changing protocols and storage. This allows it to adapt to the environment that it is running on.

The architectures presented in (Microsoft, 2016) and (Fox et al., 2012) have been shown to be effective architectures that this system incorporates at its heart. However, both lacked the necessary decoupling of the messaging and storage capabilities to achieve a flexible system. Camp IoT also incorporates at its heart, the segmentation of different states of the sensor and its functions into separate microstates and functionalities (in the forms of variables and functions). This can allow, appliance logic to be set in the cloud, and thus achieving a system where "firmware" can be constantly updated and improved over the cloud without needing to physically write the firmware to the appliance's ROM. This is also in accordance with Amazon's "*thing*" concept (Basha, Jilani, & Arun, 2016) and the thin server architecture presented in (Kovatsch et al., 2012).

Inversion of Control which is at the heart of the plugin concept, has also been confirmed as a pattern that truly allows a system to become modular, which allowed the easily plug and play mechanism of different plugins for messaging and storage. Having a language that is able to operate on different platforms and with a huge ecosystem, also proved very valuable. The large amount of libraries and packages that are available to NodeJS helped add to the success of this project.

In conclusion, the project pushes the boundaries of current IoT systems and demonstrates how adding an extra layer of abstraction and keeping the system as modular as possible can benefit users who wish to create smart areas while taking advantage of the best features from different systems, while also maximising the reduction of expenditure. This project managed to keep the system running without incurring extra costs[7] and this was achieved despite Azure's IoT infrastructure, Document DB and Event Hub, lacking a free version. However, by taking advantage of different providers i.e. not being vendor-locked, the system could make use of free tier versions of MongoDB for storage, free tier versions

[7] Of course one needs to take into consideration that there weren't a lot of users and sensors registered with the system.

RabbitMQ (via CloudAMQP) and a free version of MQTT (via CloudMQTT). Thus demonstrating that freeing the burden of satisfying vendor requirements can create more powerful and cost effective systems.

4.4 Personal Reflection

This project was great for my personal development on a plethora of different levels. I was introduced to the MEAN (MongoDB ExpressJs AngularJs NodeJs) stack which certainly highlighted the positive implications of having an all JavaScript approach (from database to client). The project included a multitude of different disciplines and techniques, each better suited at different levels. The latter include UI and client side challenges, REST API and backend challenges and some experience in distributed computing challenges. The project introduced me to RabbitMQ and other message queueing technologies that are being used more and more in large companies where systems span across multiple other subsystems and MQ helps keep the systems decoupled yet efficient.

Most importantly, the project introduced me to IoT, which is an area that has my interest although I never studied it formally. IoT is certainly the future of cities, healthcare, utility services and home automation amongst others. I also believe that it is very dangerous to be locked-in to a particular vendor or provider. I believe this system or project should either be the bases of other innovations built upon it, or at the very least it should provide inspiration for other projects of the sort, since this would certainly benefit the world of IoT.

Although this project is meant to be deployed on the cloud, it is not limited to that. This project can be configured in three ways. It could be setup in a household or a smart area on private servers or private cloud, where the access will be limited to a LAN. This would improve security but the ability to control and monitor sensors across the internet would be lost. The second configuration would be to host it publically but restrict access to certain users only (this is the presented setup). The last setup would be to host it publically and allow users to register an account, register their devices and manage them. The latter can be considered as a SaaS and would be a great tool for individuals or companies to setup up an IoT system with respect to particular smart "area" without the user going into the hassle of setting up the infrastructure and without the hassle of choosing a particular brand of devices.

4.5 Future Work

This system has been successful in its implementation, however, much more work remains to be done. One of the most prominent features that would greatly increase the value of this project would be to include Automated Rules. Rules were in the initial requirements of this project; however, they were not implemented because priority was given to features that actually contributed to overcoming the challenge of this project. Rules would incorporate certain functionalities to be triggered when certain variables reach some desired state. It could be possible for rules to span across different sensors.

Another desirable feature would be to have SDKs for the popular boards that are being used for IoT, including the Raspberry Pi, Arduino and the Particle Photon. Having these SDKs can make it possible for the messaging bus to be completely abstracted away. The end user of the SDKs would not need to worry about what messaging protocol is being used because this would be abstracted by the SDK. However, an SDK needs to be written per messaging protocol per board, which makes it quite a daunting task.

The pinnacle of abstraction would be to create an internal messaging bus, which would support multiple messaging protocols at the same time. This would enable the system to be heterogeneous and truly allow any device to be added at run-time, irrespective of which protocol it is using and hence, it can be used using said protocol.

The features mentioned above are among the most important and prominent features that would greatly improve the system. These are features whose priority would be at the top for whoever wants to improve on this project.

Bibliography

Armbrust, M., Stoica, I., Zaharia, M., Fox, A., Griffith, R., Joseph, A. D., … Rabkin, A. (2010). A view of cloud computing. *Communications of the ACM*, *53*(4), 50. http://doi.org/10.1145/1721654.1721672

Baronti, P., Pillai, P., Chook, V. W. C., Chessa, S., Gotta, A., & Hu, Y. F. (2007). Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Computer Communications*, *30*(7), 1655–1695. http://doi.org/10.1016/j.comcom.2006.12.020

Basha, S. N., Jilani, S. A. K., & Arun, S. (2016). An Intelligent Door System using Raspberry Pi and Amazon Web Services IoT, *33*(2), 84–89.

Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Communications of the ACM*, *39*(2).

Bierman G, Abadi M, T. M. (2014). Understanding TypeScript. In *Object Oriented Programming* (p. pp: 257-281). Publisher: Springer Berlin Heidelberg. http://doi.org/10.1007/978-3-662-44202-9_11

Botta, A., de Donato, W., Persico, V., & Pescapé, A. (2014). Integration of Cloud computing and Internet of Things: A survey. *Future Generation Computer Systems*, *56*, 684–700. http://doi.org/10.1016/j.future.2015.09.021

Chaniotis, I. K., Kyriakou, K.-I. D., & Tselikas, N. D. (2015). Is Node.js a viable option for building modern web applications? A performance evaluation study. *Computing*, *97*(10), 1023–1044. http://doi.org/10.1007/s00607-014-0394-9

Chappell, D. (2015). NoSQL on Microsoft Azure: An introduction. Retrieved from https://channel9.msdn.com/Events/Ignite/2015/BRK2555

Contentful Team. (n.d.). Contentful: a developer-friendly, API-first CMS. Retrieved August 21, 2016, from https://www.contentful.com/

Doukas, C., & Maglogiannis, I. (2012). Bringing IoT and Cloud Computing towards Pervasive Healthcare. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (pp. 922–926). IEEE. http://doi.org/10.1109/IMIS.2012.26

Fowler, M. (2005). Fluent Interface. Retrieved from http://martinfowler.com/bliki/FluentInterface.html

Fox, G. C., Kamburugamuve, S., & Hartman, R. D. (2012). Architecture and measured characteristics of a cloud based internet of things. In *2012 International Conference on Collaboration Technologies and Systems (CTS)* (pp. 6–12). IEEE. http://doi.org/10.1109/CTS.2012.6261020

Garfinkel, S. L. (2007). *An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS Simson.*

Grozev, N., & Buyya, R. (2014). Inter-Cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, *44*(3), 369–390. http://doi.org/10.1002/spe.2168

Hohpe, G., & Wolf, B. (2011). *Enterprise Integration Patterns.* Addison-Wsley.

Knol, A. (2013). *Dependency Injection with AngularJS*. Retrieved from https://books.google.com/books?id=9dtdAgAAQBAJ&pgis=1

Korkmaz, I., Metin, S. K., Gurek, A., Gur, C., Gurakin, C., & Akdeniz, M. (2015). A cloud based and Android supported scalable home automation system. *Computers & Electrical Engineering*, *43*, 112–128. http://doi.org/10.1016/j.compeleceng.2014.11.010

Kovatsch, M., Mayer, S., & Ostermaier, B. (2012). Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing* (pp. 751–756). IEEE. http://doi.org/10.1109/IMIS.2012.104

Krusche, S., & Alperowitz, L. (2014a). Introduction of continuous delivery in multi-customer project courses. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014* (pp. 335–343). New York, New York, USA: ACM Press. http://doi.org/10.1145/2591062.2591163

Krusche, S., & Alperowitz, L. (2014b). Introduction of continuous delivery in multi-customer project courses. *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, 335–343. http://doi.org/10.1145/2591062.2591163

Le Guilly, T., Olsen, P., Ravn, A. P., Rosenkilde, J. B., & Skou, A. (2013). HomePort: Middleware for heterogeneous home automation networks. *2013 IEEE International Conference on Pervasive Computing and Communications Workshops, PerCom Workshops 2013*, (March), 627–633. http://doi.org/10.1109/PerComW.2013.6529570

Ma, H.-D. (2011). Internet of Things: Objectives and Scientific Challenges. *Journal of Computer Science and Technology*, *26*(6), 919–924. http://doi.org/10.1007/s11390-011-1189-5

Maksimović, M., Vujović, V., Davidović, N., Milošević, V., & Perišić, B. (2014). Raspberry Pi as Internet of Things hardware : Performances and Constraints. *Design Issues*, *3*(JUNE), 8.

Martin Fowler. (2004). Inversion of Control Containers and the Dependency Injection pattern. Retrieved July 24, 2016, from http://martinfowler.com/articles/injection.html#ServiceLocatorVsDependencyInjection

Microsoft. (2016). Microsoft Azure IoT Reference Architecture. Microsoft Corporation.

Mineraud, J., Mazhelis, O., Su, X., & Tarkoma, S. (2016). A gap analysis of Internet-of-Things platforms. *Computer Communications*, *89*, 5–16. http://doi.org/10.1016/j.comcom.2016.03.015

MQTT V3.1 Protocol Specification. (n.d.). Retrieved from http://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

Olawale, O., & Brett, F. (2016). *Core Tenets of IoT*.

Onion. (2016). Cloud – Onion. Retrieved August 28, 2016, from https://onion.io/cloud/

Particle. (2016). Particle Guides | Particle Dev. Retrieved August 28, 2016, from https://docs.particle.io/guide/tools-and-features/dev/#cloud-variables-amp-functions

Pühringer, C. (n.d.). *Cloud Computing for Home Automation*. Vienna University of Technology.

Satzger, B., Hummer, W., Inzinger, C., Leitner, P., & Dustdar, S. (2013). Winds of change: From vendor lock-in to the meta cloud. *IEEE Internet Computing*, *17*(1), 69–73. http://doi.org/10.1109/MIC.2013.19

Shelby, Z., Hartke, K., & Bormann, C. (2014). The Constrained Application Protocol (CoAP). *Rfc 7252*, 112. http://doi.org/10.1007/s13398-014-0173-7.2

Simmhan, Y., Kumbhare, A. G., Cao, B., & Prasanna, V. (2011). An Analysis of Security and Privacy Issues in Smart Grid Software Architectures on Clouds. In *2011 IEEE 4th International Conference on Cloud Computing* (pp. 582–589). IEEE. http://doi.org/10.1109/CLOUD.2011.107

Videla, A., & Williams, J. J. W. (2012). *RabbitMQ in action distributed messaging for everyone*. Manning.

Vinoski, S. (2006). Advanced Message Queuing Protocol. *IEEE Internet Computing*, *10*(6), 87–89. http://doi.org/10.1109/MIC.2006.116

Vujovi, V., & Maksimovi, M. (2014). Raspberry Pi as a Sensor Web node for home automation. *Computers and Electrical Engineering*, *44*, 153–171. http://doi.org/10.1016/j.compeleceng.2015.01.019

Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, *1*(1), 7–18. http://doi.org/10.1007/s13174-010-0007-6

Appendix

Appendix A: Bulleted Challenges Aims and Objectives

Challenges

The challenge of this project is to tackle the problem of heterogeneity i.e. vendor lock-in in the world of IoT. This is split into further sub-challenges:

- Sensor Communication Protocol & Storage lock in.
- Operating system/Hardware lock-in.

Aims

- To build a cloud based Service Oriented Architecture that can manages devices (sensors) and the communication between them, together with storing data received from devices.
- To make the SOA a system that can be used across various platforms and cloud providers while making sure that it takes advantage of the platforms capabilities as much as possible.
- To make the service generic enough to accommodate a large variety of devices (sensors) and clients.
- To Build a cross-platform Dashboard in order to provide a user-friendly GUI to manage devices.

Objectives

- Build a REST, web standards compliant SOA to make sure that most clients can connect to it and gain access to sensor data and sensor management.

- Use IoC to create an abstraction layer that accepts different plugins to allow the system to communicate with the underlying platform.

- Build at least a plugin for Messaging, which implement the platform specific features that are needed from the platform.

- Build a client using web technologies that can be packaged to run on other platforms including iOS and Android.

Appendix B: Requirements Obsolete or not Implemented

ID	001
FUNCTION	User Registration
TYPE	Functional
PRIORITY	MUST
DESTINATION	IoT Cloud Controller/Dashboard API
DESCRIPTION	A user may should be able to supply personal information to the system in order to register him/herself with the system.
FITNESS CRITERION	A REST endpoint under the resource */Dahsboard/User/Register* should be created. Users can create POST requests with their email address and password. A new user should be added in the database.
DEPENDS ON	009

ID	002
FUNCTION	Smart Area Registration
TYPE	Functional
PRIORITY	COULD

DESTINATION	IoT Cloud Controller/Dashboard API
DESCRIPTION	A user should be able to register multiple Smart Areas (homes or otherwise). Each sensor shall be associated with a Smart Area. This will serve as a logical partitioning of the sensors. It can also be utilised as a means to achieve more flexible user access control.
FITNESS CRITERION	A REST endpoint under the resource */Dahsboard/User/SmartArea/Register* should be created. Users can create POST requests with an alias representing the name of the Area. A new Smart Area should be added in the database.
DEPENDS ON	010

ID	010
FUNCTION	Smart Area Persistence
TYPE	Functional
PRIORITY	COULD
DESTINATION	IoT Cloud Controller/Storage API
DESCRIPTION	CRUD operations for managing Smart Areas
FITNESS CRITERION	The system must be able to create, retrieve update and delete smart area data on cloud storage.
DEPENDS ON	013, 014

ID	014
FUNCTION	Amazon Web Services Storage Plugin
TYPE	Domain
PRIORITY	SHOULD
DESTINATION	IoT Cloud Controller/Plugin SDK
DESCRIPTION	A plugin that provides the storage capabilities required by this system and abstracts the integration to the AWS cloud Storage
FITNESS CRITERION	CRUD operations can be performed on AWS cloud storage.
DEPENDS ON	008

ID	016
FUNCTION	CoAp server plugin for CoAp enabled Devices
TYPE	Domain
PRIORITY	WON'T
DESTINATION	IoT Cloud Controller/Messaging SDK/Central Messaging Unit
DESCRIPTION	The Plugin SDK and Messaging API should enable multiple components to be for messaging purposes. A CoAp Server

	plugin should be developed to enable communication between the IoT Cloud Controller and CoAp enabled sensors.
FITNESS CRITERION	The system should be able to send messages and receive messages via the CoAp protocol
DEPENDS ON	008
ID	019
FUNCTION	Rule Creation
TYPE	Functional
PRIORITY	COULD
DESTINATION	IoT Cloud Controller/RuleAPI
DESCRIPTION	The system should allow for cloud rules to be specified in the cloud. Rules should make use of one or more *streams* and *jobs* to automate some real job in the real world.
FITNESS CRITERION	A REST endpoint *Rule/register* should be created. By POSTing the name, streams required, *jobs* required and the rule script, the user can call the end point and register the rule. The rule should be persisted in cloud storage.
DEPENDS ON	020

ID	020

FUNCTION	Rule Persistence
TYPE	Functional
PRIORITY	SHOULD
DESTINATION	IoT Cloud Controller/StorageAPI
DESCRIPTION	CRUD operations for managing Rules
FITNESS CRITERION	The system must be able to create, retrieve update and delete rules on cloud storage.
DEPENDS ON	013, 014

ID	021
FUNCTION	Rule Retrieval
TYPE	Functional
PRIORITY	COULD
DESTINATION	IoT Cloud Controller/RuleAPI
DESCRIPTION	Ability to load the retrieve a list of rules
FITNESS CRITERION	A REST endpoint /*Rule*/*list* will retrieve all the rules data.

DEPENDS ON	020

ID	022
FUNCTION	User Registration
TYPE	Functional
PRIORITY	MUST
DESTINATION	Dashboard/Users
DESCRIPTION	A registration screen that allows users to input their email, password and a password confirmation to register themselves with the system.
FITNESS CRITERION	The client application should send a POST request on the provided endpoint with the aforementioned data in the requests payload.
DEPENDS ON	001

ID	029
FUNCTION	Rule Maintenance Screen
TYPE	Functional
PRIORITY	SHOULD
DESTINATION	Dashboard/Rule
DESCRIPTION	A screen shall be presented to the users so that they can specify the name of rule, the variables it requires, the job it is able to invoke and the actual rule script.
FITNESS CRITERION	The client application should send a POST request on the provided endpoint with the aforementioned data so that a rule can be registered. In the case of an existing rule, a PUT request shall be issued to update the rule data.
DEPENDS ON	019

Appendix C: Code Listing

Please note that the code snippets listed in this section, are not complete. These are the snippets most important to solving the problems and challenges that this project is trying to fix.

Custom IoC Container

The following are the classes that make up the custom IoC container that is at the heart of this system and makes it possible to achieve a decoupled system via the dependency injection and IoC principles, that subsequently make the plugin architecture possible.

```
import {ContainerItem} from "./ioc.model";


export class ContainerService {
    private container = new Map<string, ContainerItem>();

    register(item: ContainerItem): void {
        if (!this.container.has(item.key)) {
            this.container.set(item.key, item);
        }
    }

    getByKey<T>(key: string): T {
        let item = this.tryGet(key);
        if (this.shouldInstantiate(item)) {
            throw `${key} is not a singleton and it should be created. Try
using the create function instead.`;
        }

        return item.item;
    }

    create<T>(instantiatee: {new(...args: any[]): T; }): T {
        let injections = (<any>instantiatee).$inject;
        let items: any[];
        if (injections) {
            items = this.resolve(injections);
        }

        const _instantiatee = instantiatee;
        let ins = function (...args: any[]) {
            return new _instantiatee(...args);
        };
        ins.prototype = _instantiatee;

        ins = ins.apply(null, items);
        return <T><any>ins;
    }

    private resolve(injectionKeys: string[]): any[] {
        let foundItems: any[] = [];

        injectionKeys.forEach((key: string) => {
            let containerItem = this.tryGet(key);
            let instance = containerItem.item;

            if (this.shouldInstantiate(containerItem)) {
                instance = this.create(instance);

                if (containerItem.isSingleton) {
                    containerItem.asNonInstantiatable();
                    containerItem.item = instance;
                }
            }

            foundItems.push(instance);
        });

        return foundItems;
    }

    private tryGet(key: string): ContainerItem {
        let containerItem = this.container.get(key);
        if (containerItem) {
            return containerItem;
        }

        throw `Dependency ${key} not registered!`;
    }

    private shouldInstantiate(item: ContainerItem): boolean {
        let nonSingleton = !item.isSingleton && item.isInstantiatable;
        let lazySingleton = item.isSingleton && item.isInstantiatable;

        return nonSingleton || lazySingleton;
    }
}
```

```
import {ContainerService} from "./ioc.service";

export class ContainerItem {
    private _isSingleton = false;
    public key: string;
    public item: any;
    private _isInstantiatable = true;

    constructor(private container: ContainerService) {
    }

    get isSingleton(): boolean {
        return this._isSingleton;
    }

    get isInstantiatable(): boolean {
        return this._isInstantiatable;
    }

    forKey(key: string): ContainerItem {
        this.key = key;
        return this;
    }

    use(item: Object|Function): ContainerItem {
        this.item = item;
        return this;
    }

    asSingleton(): ContainerItem {
        this._isSingleton = true;
        this._isInstantiatable = false;
        this.item = this.container.create(this.item);

        return this;
    }

    asLazySingleton(): ContainerItem {
        this._isSingleton = true;
        return this;
    }

    asNonInstantiatable(): ContainerItem {
        this._isInstantiatable = false;
        return this;
    }

    andRegister(): void {
        this.container.register(this);
    }

    static create(container: ContainerService): ContainerItem {
        return new ContainerItem(container);
    }
}


export function setupBrokerContainer(): ContainerService {
    "use strict";

    let container = new ContainerService();
```

```
ContainerItem.create(container)
    .forKey("Config")
    .use(Config)
    .asSingleton()
    .andRegister();

ContainerItem.create(container)
    .forKey("Logger")
    .use(setUpLogger())
    .asNonInstantiatable()
    .andRegister();

ContainerItem.create(container)
    .forKey("IBrokerConfig")
    //.use(AmqpConfig)
    .use(MqttConfig)
    .andRegister();

ContainerItem.create(container)
    .forKey("IBrokerService")
    //.use(AmqpBroker)
    .use(MqttBroker)
    .andRegister();

ContainerItem.create(container)
    .forKey("ITaskService")
    .use(TaskService)
    .asSingleton()
    .andRegister();

ContainerItem.create(container)
    .forKey("IVariableTaskProvider")
    .use(VariableTaskProvider)
    .andRegister();

ContainerItem.create(container)
    .forKey("IStreamService")
    .use(StreamService)
    .andRegister();

ContainerItem.create(container)
    .forKey("IStreamRepository")
    .use(StreamRepository)
    .andRegister();

ContainerItem.create(container)
    .forKey("IValidator")
    .use(Validator)
    .andRegister();

ContainerItem.create(container)
    .forKey("IVariableService")
    .use(VariableService)
    .andRegister();

ContainerItem.create(container)
    .forKey("IVariableModelRepository")
    .use(VariableRepository)
    .andRegister();
    ContainerItem.create(container)
        .forKey("DocumentDbConfig")
        //.use(container.getByKey<Config>("Config").mongoDbAzureSetup)
        //.use(container.getByKey<Config>("Config").mongoDbLocalSetup)
        .use(container.getByKey<Config>("Config").mongoDbAmazonSetup)
        .asNonInstantiatable()
        .andRegister();

    ContainerItem.create(container)
        .forKey("IStorageSetup")
        .use(Setup)
        .asSingleton()
        .andRegister();

    return container;
}
```

Plugin Contracts (Storage & Messaging)

The following are the interfaces that need to be shared with and implemented by plugins to ensure that the required functionality by the core code is provided:

```
import {Task} from "./task.model";

export interface IBrokerService {
    publish(sensorId: string, topicId: string, data: any): Promise<void>;
    registerSensor(sensorId: string): Promise<any>;
    registerJob(sensorId: string, topicId: string): Promise<any>;
    subscribe(sensorId: string, topicId: string, task: Task,
isHandlerRemovable?: boolean): Promise<string>;
    unsubscribe(handlerId: string): Promise<void>;
    cleanUp(hard?: boolean): Promise<void>;
}

export interface IBrokerConfig {
    configure(): Promise<any>;
}

export interface BrokerConfig {
    user: string;
    password: string;
    host: string;
    port: number;
    vhost?: string;
    protocol: string;
}

import {User} from "../user/user";
import {Sensor} from "../sensor/sensor";
import {CampModel} from "../shared/model/common.model";
import {Variable} from "../variable/variable";
import {Stream} from "../stream/stream";
import {Job} from "../job/job.model";

export interface ICampModelRepository<T extends CampModel> {
    save(model: T): Promise<T>;
    update(id: string, updated: Object): Promise<T>;
    findMany(ids: string[]): Promise<T[]>;
    delete(id: string): Promise<void>;
    find(id: string): Promise<T>;
    deleteMany(ids: string[]): Promise<void[]>;
}

export interface IUserModelRepository extends ICampModelRepository<User> {
    findByEmail(email: string): Promise<User>;
    addSensor(id: string, sensorsIds: string[]): Promise<User>;
    findUserStubByEmail(email: string): Promise<User>;
    deleteSensor(id: string, sensorId: string): Promise<User>;
}

export interface ISensorModelRepository extends ICampModelRepository<Sensor>
{
    addVariable(id: string, variableIds: string[]): Promise<Sensor>;
    deleteVariable(id: string, variableId: string): Promise<Sensor>;
    addJob(sensorId: string, jobIds: string[]): Promise<Sensor>;
    deleteJob(sensorId: string, jobId: string): Promise<Sensor>;
}

export interface IVariableModelRepository extends
ICampModelRepository<Variable> {
    updateValue(variableId: string, value: number|boolean):
Promise<Variable>;
}

export interface IStreamRepository extends ICampModelRepository<Stream> {
    deleteByVariableId(variableId: any): Promise<Stream[]>;
    getStreamsByVariableId(variableId: string, limit?: number):
Promise<Stream[]>;
}

export interface IJobRepository extends ICampModelRepository<Job> {
}
export interface DocumentDbConfig {
    endpoint: string;
    key?: string;
    operationalDbName: string;
}

export interface IStorageSetup {
    initConnection(): Promise<DbConnectionResult>;
}

export interface DbConnectionResult {
    db: any;
    dbName: string;
}

export interface StorageError extends Error {
    code: number;
    message: string;
}
```

Concrete Implementation of Plugins (MongoDb, RabbitMq [AMQP], MQTT)

The following is the code listing for the implementation of plugins that do the actual logic to interact with the database (MongoDb), with an AMQP broker (using RabbitMQ) and an MQTT broker.

```
import {MongooseModelService} from "./model.repository";
import {IJobRepository} from "../../../storage/storage.service";
import {Job} from "../../../job/job";
import * as mongoose from "mongoose";
import {Schema} from "mongoose";

export class JobRepository extends MongooseModelService<Job> implements
IJobRepository {

    private static modelDefinition = mongoose.model("Job", new Schema({
        name: String,
        parameters: [Schema.Types.String]
    }));

    constructor() {
        super(JobRepository.modelDefinition);
    }

    save(model: Job): Promise<Job> {
        return this.objectify(this.saveModel(model));
    }

    update(id: string, updated: Object): Promise<Job> {
        return undefined;
    }

    findMany(ids: string[]): Promise<Job[]> {
        return this.objectifyMany(
            <any>this.modelDefinition
                .find({
                    _id: {$in: ids}
                })
                .select({__v: 0})
                .exec()
        );
    }

    delete(id: string): Promise<void> {
        return this.remove(id);
    }

    find(id: string): Promise<Job> {
        return this.objectify(
            <any>this.modelDefinition
                .findOne({
                    _id: id
                })
                .select({__v: 0})
                .exec()
        );
    }
    deleteMany(ids: string[]): Promise<void[]> {
        return this.removeMany(ids);
    }
}
```

```
import {StorageError} from "../../../storage/storage";
import * as mongoose from "mongoose";
import {CampModel} from "../../../shared/model/common.model";
import {WriteError} from "mongodb";
import {Model, Document, Types} from "mongoose";

export abstract class MongooseModelService<T extends CampModel> {

    constructor(protected modelDefinition: Model<any>) {
        mongoose.Promise = <any>global.Promise;
    }

    protected saveModel(model: T): Promise<Document> {
        let generatedModel = new this.modelDefinition(model);

        return (<any>generatedModel).save()
            .catch((err: WriteError) => {
                let error: StorageError = {
                    code: err.code,
                    message: err.errmsg || (<any>err).message,
                    name: "Save"
                };

                return Promise.reject(error);
            });
    }

    protected findManyModel(queryObject: any): Promise<Document[]> {
        return <any>this.modelDefinition.find(queryObject).exec();
    }

    protected findOne(queryObject: any): Promise<Document> {
        return <any>this.modelDefinition.findOne(queryObject).exec();
    }

    protected updateModel(id: string, updated: any): Promise<Document> {
        return (<any>this.modelDefinition.findOneAndUpdate({_id: id},
updated)
            .exec())
            .catch((err: WriteError) => {
                return Promise.reject({
                    code: err.code,
                    message: err.errmsg,
                    name: "Update"
                });
            });
    }
    protected remove(id: string): Promise<void> {
        return (<any>this.modelDefinition.findOneAndRemove({_id: id})
            .exec())
            .catch((err: any) => {
                return Promise.reject({
                    code: err.code,
                    message: err.errmsg ? err.errmsg : err.message,
                    name: "Remove"
                });
            });
    }

    protected removeMany(ids: string[]): Promise<void[]> {
        return (<any>this.modelDefinition
            .find({_id: { $in: ids }})
            .remove()
            .exec())
            .catch((err: any) => {
                return Promise.reject({
                    code: err.code,
                    message: err.errmsg ? err.errmsg : err.message,
                    name: "RemoveMany"
                });
            });
    }

    protected objectify(promise: Promise<Document>): Promise<T> {
        return promise
            .then((dbModel: Document) => {
                return <any>this.convertToRawObject(dbModel);
            })
            .catch((err: any) => {
                return Promise.reject({
                    code: err.code,
                    write_message: err.errmsg,
                    message: err.message
                });
            });
    }

    protected objectifyMany(promise: Promise<Document[]>): Promise<T[]> {
        return promise
            .then((dbModels: Document[]) => {
                let models: T[] = [];

                dbModels.forEach((dbModel: Document) => {
                    let raw = this.convertToRawObject(dbModel);
                    models.push(raw);
                });

                return <any>models;
            })
            .catch((err: WriteError) => {
                return Promise.reject({
                    code: err.code,
                    message: err.errmsg
                });
            });
    }
    private convertToRawObject(dbModel: Document): T {
        if (!dbModel) {
            return null;
        }

        let raw = <T>dbModel.toObject();
        raw._id = raw._id.toHexString();

        return raw;
    }

    static toMongooseId(ids: string[]): Types.ObjectId[] {
        let objectIds: Types.ObjectId[] = [];
        ids.forEach((id: string) => {
            let objectId = Types.ObjectId.createFromHexString(id);
            objectIds.push(objectId);
        });

        return objectIds;
    }
}
```

```
import {IStorageSetup, DocumentDbConfig, DbConnectionResult} from
"../../../storage/storage";
import * as mongoose from "mongoose";
import {Logger} from "bunyan";

export class Setup implements IStorageSetup {
    static $inject = ["DocumentDbConfig", "Logger"];

    constructor(private config: DocumentDbConfig, private logger: Logger) {
    }

    initConnection(): Promise<DbConnectionResult> {
        return new Promise<DbConnectionResult>((resolve: Function, reject:
Function) => {
            try {
                let db = mongoose.connect(this.config.endpoint);
                db.connection.on("error", (err: Error) => {
                    this.logger.error(err);
                });

                resolve({
                    db: db,
                    dbName: "MongoDb"
                });
            } catch (err) {
                reject(err);
            }
        });
    }
}
```

```
import {MongooseModelService} from "./model.repository";
import {ISensorModelRepository} from "../../../storage/storage.service";
import {Sensor} from "../../../sensor/sensor";
import {Schema} from "mongoose";
import * as mongoose from "mongoose";

export class SensorRepository extends MongooseModelService<Sensor>
implements ISensorModelRepository {

    private static modelDefinition = mongoose.model("Sensor", new Schema({
        name: String,
        variables: [Schema.Types.ObjectId],
        jobs: [Schema.Types.ObjectId]
    }));

    constructor() {
        super(SensorRepository.modelDefinition);
    }

    save(sensor: Sensor): Promise<Sensor> {
        return this.objectify(this.saveModel(sensor));
    }

    update(id: string, updated: Object): Promise<Sensor> {
        return this.objectify(this.updateModel(id, updated));
    }

    delete(id: string): Promise<void> {
        return this.remove(id);
    }

    findMany(sensorIds: string[]): Promise<Sensor[]> {
        return this.objectifyMany(
            <any>this.modelDefinition
                .find({
                    _id: {$in: sensorIds}
                })
                .select({__v: 0, jobs: 0, variables: 0})
                .exec()
        );
    }

    find(sensorId: string): Promise<Sensor> {
        return this.objectify(
            <any>this.modelDefinition
                .findOne({
                    _id: sensorId
                })
                .select({__v: 0})
                .exec()
        );
    }

    addVariable(id: string, variableIds: string[]): Promise<Sensor> {
        let ids = MongooseModelService.toMongooseId(variableIds);
        return this.objectify(
            <any>this.modelDefinition.findOneAndUpdate(
                {_id: id},
                {
                    $push: {variables: ids}
                },
                {
                    upsert: true
                }
            ).exec()
        );
    }

    deleteVariable(id: string, variableId: string): Promise<Sensor> {
        let ids = MongooseModelService.toMongooseId([variableId]);
        return this.objectify(
            <any>this.modelDefinition.findOneAndUpdate(
                {_id: id},
                {
                    $pull: {variables: ids}
                },
                {
                    upsert: true
                }
            ).exec()
        );
    }

    deleteMany(ids: string[]): Promise<void[]> {
        return undefined;
    }

    addJob(sensorId: string, jobIds: string[]): Promise<Sensor> {
        let ids = MongooseModelService.toMongooseId(jobIds);
        return this.objectify(
            <any>this.modelDefinition.findOneAndUpdate(
                {_id: sensorId},
                {
                    $push: {jobs: ids}
                },
                {
                    upsert: true
                }
            ).exec()
        );
    }

    deleteJob(sensorId: string, jobId: string): Promise<Sensor> {
        let ids = MongooseModelService.toMongooseId([jobId]);
        return this.objectify(
            <any>this.modelDefinition.findOneAndUpdate(
                {_id: sensorId},
                {
                    $pull: {jobs: ids}
                },
                {
                    upsert: true
                }
            ).exec()
        );
    }
}
```

```
import {MongooseModelService} from "./model.repository";
import {Stream} from "../../../stream/stream";
import {IStreamRepository} from "../../../storage/storage.service";
import {Schema} from "mongoose";
import * as mongoose from "mongoose";

export class StreamRepository extends MongooseModelService<Stream>
implements IStreamRepository {

    private static modelDefinition = mongoose.model("Stream", new Schema({
        value: mongoose.Schema.Types.Mixed,
        variableId: {type: Schema.Types.ObjectId, index: true},
        timeStamp: Date
    }));

    constructor() {
        super(StreamRepository.modelDefinition);
    }

    save(stream: Stream): Promise<Stream> {
        return this.objectify(this.saveModel(stream));
    }

    update(id: string, updated: Object): Promise<Stream> {
        return undefined;
    }

    findMany(ids: string[]): Promise<Stream[]> {
        return undefined;
    }

    delete(id: string): Promise<void> {
        return undefined;
    }

    find(id: string): Promise<Stream> {
        return undefined;
    }

    deleteByVariableId(variableId: any): Promise<Stream[]> {
        return (<any>this.modelDefinition
            .find({variableId: variableId})
            .remove()
            .exec())
            .catch((err: any) => {
                return Promise.reject({
                    code: err.code,
                    message: err.errmsg ? err.errmsg : err.message,
                    name: "Remove"
                });
            });
    }

    deleteMany(ids: string[]): Promise<void[]> {
        return undefined;
    }


    getStreamsByVariableId(variableId: string, limit: number = 0):
Promise<Stream[]> {
        return this.objectifyMany(<any>this.modelDefinition
            .find({variableId: variableId})
            .sort([["timeStamp", "descending"]])
            .limit(limit)
            .select({__v: 0})
            .exec());
    }
}
```

```
import {MongooseModelService} from "./model.repository";
import {User} from "../../../user/user";
import {IUserModelRepository} from "../../../storage/storage";
import {Schema} from "mongoose";
import * as mongoose from "mongoose";

export class UserRepository extends MongooseModelService<User> implements
IUserModelRepository {

    private static modelDefinition = mongoose.model("User", new Schema({
        email: {type: String, unique: true},
        name: String,
        hashedPass: String,
        sensors: [Schema.Types.ObjectId]
    }));

    constructor() {
        super(UserRepository.modelDefinition);
    }

    find(id: string): Promise<User> {
        return this.objectify(this.findOne({_id: id}));
    }

    findByEmail(email: string): Promise<User> {
        return this.objectify(this.findOne({email: email}));
    }

    findUserStubByEmail(email: string): Promise<User> {
        return this.objectify(
            <any>this.modelDefinition.findOne({
                    email: email
                })
                .select({sensors: 0, __v: 0})
                .exec()
        );
    }

    save(user: User): Promise<User> {
        return this.objectify(this.saveModel(user));
    }

    addSensor(id: string, sensorsIds: string[]): Promise<User> {
        let ids = MongooseModelService.toMongooseId(sensorsIds);
        return this.objectify(
            <any>this.modelDefinition.findOneAndUpdate(
                {_id: id},
                {
                    $push: {sensors: ids}
                },
                {
                    upsert: true
                }
            ).exec()
        );
    }

    deleteSensor(id: string, sensorId: string): Promise<User> {
        let ids = MongooseModelService.toMongooseId([sensorId]);
        return this.objectify(
            <any>this.modelDefinition.findOneAndUpdate(
                {_id: id},
                {
                    $pull: {sensors: ids}
                },
                {
                    upsert: true
                }
            ).exec()
        );
    }

    update(id: string, updated: Object): Promise<User> {
        return undefined;
    }

    findMany(ids: string[]): Promise<User[]> {
        return undefined;
    }

    delete(id: string): Promise<void> {
        return undefined;
    }

    deleteMany(ids: string[]): Promise<void[]> {
        return undefined;
    }
}
```

```
import {MongooseModelService} from "./model.repository";
import {IVariableModelRepository} from "../../../storage/storage.service";
import {Schema} from "mongoose";
import * as mongoose from "mongoose";
import {Variable} from "../../../variable/variable";

export class VariableRepository extends MongooseModelService<Variable>
implements IVariableModelRepository {

    private static modelDefinition = mongoose.model("Variable", new Schema({
        name: String,
        type: {type: Number, default: 0, min: 0, max: 1},
        value: {type: mongoose.Schema.Types.Mixed, default: 0, required:
false },
        isStream: {type: Boolean, default: false}
    }));

    constructor() {
        super(VariableRepository.modelDefinition);
    }

    save(model: Variable): Promise<Variable> {
        return this.objectify(this.saveModel(model));
    }

    update(id: string, updated: Object): Promise<Variable> {
        return undefined;
    }

    findMany(ids: string[]): Promise<Variable[]> {
        return this.objectifyMany(
            <any>this.modelDefinition
                .find({
                    _id: {$in: ids}
                })
                .select({__v: 0})
                .exec()
        );
    }

    delete(id: string): Promise<void> {
        return this.remove(id);
    }

    deleteMany(ids: string[]): Promise<void[]> {
        return this.removeMany(ids);
    }

    find(id: string): Promise<Variable> {
        return this.objectify(
            <any>this.modelDefinition
                .findOne({
                    _id: id
                })
                .select({__v: 0})
                .exec()
        );
    }

    updateValue(variableId: string, value: number|boolean):
Promise<Variable> {
        return this.objectify(
            <any>this.modelDefinition
                .findByIdAndUpdate(variableId, {
                    $set: {
                        value: value
                    }
                }).exec());
    }
}
```

```
import {IBrokerService, IBrokerConfig} from
"../../../messaging/broker.service";
import * as rabbit from "rabbot";
import {Message} from "rabbot";
import {Task} from "../../../messaging/messaging";
import * as uuid from "node-uuid";

export class AmqpBroker implements IBrokerService {

    static $inject = ["IBrokerConfig"];

    private handlers: Map<string, any>;
    private removableHandlers: Map<string, any>;

    constructor(private brokerConfig: IBrokerConfig) {

        brokerConfig
            .configure()
            .then((config: any) => {
                return rabbit.addConnection({
                    uri: config
                });
            });

        this.handlers = new Map<string, any>();
        this.removableHandlers = new Map<string, any>();
    }

    publish(sensorId: string, topicId: string, data: any): Promise<void> {
        return rabbit.publish(sensorId, {
            routingKey: topicId,
            type: topicId,
            body: data
        });
    }

    registerSensor(sensorId: string): Promise<any> {
        return rabbit.addExchange(sensorId, "topic", {
            durable: true,
            persistent: true,
            autoDelete: false
        });
    }

    registerJob(sensorId: string, topicId: string): Promise<any> {
        return this.addQueue(sensorId, topicId)
            .then(() => {
                rabbit.startSubscription(topicId);
            });
    }

    subscribe(sensorId: string, topicId: string, task: Task,
isHandlerRemovable: boolean = false): Promise<string> {
        return this
            .addQueue(sensorId, topicId)
            .then(() => {
                let guid = uuid.v4();
                return Promise.all([
                    this.initBrokerSubscription(topicId, task, guid),
                    guid
                ]);
            })
            .then((handlerData: any[]) => {
                rabbit.startSubscription(topicId);
                return handlerData;
            })
            .then((handlerData: any[]) => {
                let guid = handlerData[1];
                let handler = handlerData[0];

                let handlerList = this.handlers;
                if (isHandlerRemovable) {
                    handlerList = this.removableHandlers;
                }

                handlerList.set(guid, handler);

                return guid;
            });
    }

    unsubscribe(handlerId: string): Promise<void> {

        let unsubscriber = (resolve: Function) => {
            let handler = this.removableHandlers.get(handlerId);
            let handlerList = this.removableHandlers;

            if (!handler) {
                handler = this.handlers.get(handlerId);
                handlerList = this.handlers;
            }

            handler.remove();
            handlerList.delete(handlerId);

            resolve();
        };

        return new Promise<void>(unsubscriber);
    }

    cleanUp(hard: boolean = false): Promise<void> {

        let cleaner = (resolve: Function) => {
            let remover = (handler: any) => {
                handler.remove();
            };

            this.removableHandlers.forEach(remover);
            this.removableHandlers.clear();

            if (hard) {
                this.handlers.forEach(remover);
                this.handlers.clear();
            }

            resolve();
        };

        return new Promise<void>(cleaner);
    }
    private addQueue(sensorId: string, topicId: string): Promise<any> {
        return rabbit
            .addQueue(topicId, {
                durable: true,
                autoDelete: false,
                noAck: false,
                exclusive: false,
                subscribe: false,
            })
            .then(() => {
                return rabbit.bindQueue(sensorId, topicId, [topicId]);
            });
    }

    private initBrokerSubscription(topicId: string, task: Task, handlerId:
string): Promise<any> {
        const subscriber = (resolve: Function) => {
            rabbit.nackOnError();
            let handler = rabbit.handle(topicId, (message: Message) => {
                let parsedBody = message.body;
                if (parsedBody.toJSON) {
                    parsedBody = JSON.parse(parsedBody.toString());
                }

                parsedBody.handlerId = handlerId;

                task.task.apply(task.context, [parsedBody]);

                message.ack();
            });

            resolve(handler);
        };

        return new Promise<void>(subscriber);
    }
}
```

```
import {IBrokerConfig} from "../../../messaging/broker.service";
import {Config} from "../../../config";

export class AmqpConfig implements IBrokerConfig {
    static $inject = ["Config"];

    constructor(private config: Config) {
    }

    configure(): Promise<any> {
        let settings = this.config.brokerSetup;
        return Promise

.resolve(`amqp://${settings.user}:${settings.password}@${settings.host}/${se
ttings.vhost}`);
    }
}
```

```
import {IBrokerService, Task, IBrokerConfig} from
"../../../messaging/messaging";
import * as mqtt from "mqtt";
import {Client} from "mqtt";
import * as uuid from "node-uuid";
import * as _ from "lodash";

export class MqttBroker implements IBrokerService {

    static $inject = ["IBrokerConfig"];

    private client: Client;
    private handlers = new Map<string, HandlerContent>();
    private removableHandlers = new Map<string, HandlerContent>();

    constructor(private config: IBrokerConfig) {
        config.configure()
            .then((settings: any) => {

                this.client = mqtt.connect(settings);

                this.client.on("message", (topic: string, message: Buffer)
=> {

                    let parsedMessage = JSON.parse(message.toString());

                    let handlerList: HandlerContent[] = [];
                    let handler = this.handlers.get(topic);
                    if (handler) {
                        handlerList.push(handler);
                    }

                    let removables: HandlerContent[] = [];

                    this.removableHandlers.forEach((hndl: HandlerContent,
key: string) => {
                        let keyParts = key.split(":");
                        if (topic === keyParts[0] && hndl.handlerId ===
keyParts[1]) {

                            removables.push(hndl);
                        }
                    });

                    handlerList = _.concat(handlerList, removables);

                    _.forEach(handlerList, (hndl: HandlerContent) => {
                        parsedMessage.handlerId = hndl.handlerId;
                        hndl.task.task.apply(hndl.task.context,
[parsedMessage]);
                    });
                });
            });
    }

    publish(sensorId: string, topicId: string, data: any): Promise<void> {
        let publisher = (resolve: Function, reject: Function) => {
            try {
                this.client.publish(topicId, JSON.stringify(data));
                resolve();
            } catch (err) {
                reject(err);
            }
        };

        return new Promise<void>(publisher);
    }

    registerSensor(sensorId: string): Promise<any> {
        return Promise.resolve();
    }

    registerJob(sensorId: string, topicId: string): Promise<any> {
        let jobRegister = (resolve: Function, reject: Function) => {
            try {
                this.client.subscribe(topicId);
                resolve();
            } catch (err) {
                reject(err);
            }
        };

        return new Promise(jobRegister);
    }

    subscribe(sensorId: string,
              topicId: string,
              task: Task,
              isHandlerRemovable: boolean = false): Promise<string> {
        let variableRegister = (resolve: Function, reject: Function) => {
            try {
                this.client.subscribe(topicId);
                let guid = uuid.v4();

                let key = topicId;
                let handlerList = this.handlers;
                if (isHandlerRemovable) {
                    handlerList = this.removableHandlers;
                    key = `${topicId}:${guid}`;
                }

                handlerList.set(key, {
                    task: task,
                    handlerId: guid
                });

                resolve(guid);
            } catch (err) {
                reject(err);
            }
        };

        return new Promise(variableRegister);
    }

    unsubscribe(handlerId: string): Promise<void> {
        let unsubscriber = (resolve: Function, reject: Function) => {
            try {

                for (let handleItem of this.handlers) {
                    let handle: HandlerContent = handleItem[1];
                    if (handle.handlerId === handlerId) {
                        let topic = handleItem[0];
                        this.handlers.delete(topic);
                        this.client.unsubscribe(topic);
                        break;
                    }
                }

                this.removableHandlers.forEach((handle: HandlerContent, key:
string) => {
                    if (key.indexOf(handlerId) > -1) {
                        this.removableHandlers.delete(key);
                    }
                });

                resolve();
            } catch (err) {
                reject(err);
            }
        };

        return new Promise<void>(unsubscriber);
    }

    cleanUp(hard?: boolean): Promise<void> {
        let cleaner = (resolve: Function) => {
            this.removableHandlers.clear();

            if (hard) {
                this.handlers.clear();
            }

            resolve();
        };

        return new Promise<void>(cleaner);
    }
}

interface HandlerContent {
    task: Task;
    handlerId: string;
}

import {IBrokerConfig} from "../../../messaging/messaging";
import {Config} from "../../../config";

export class MqttConfig implements IBrokerConfig {

    static $inject = ["Config"];

    constructor(private config: Config) {
    }


    configure(): Promise<any> {
        let settings = this.config.brokerSetup;
        return Promise

.resolve(`mqtt://${settings.user}:${settings.password}@${settings.host}:${se
ttings.port}`);
    }
}
```

The Messaging Process

The following code is the process that is forked from the system's main process in order to dedicate a separate CPU core or possibly a whole other CPU to the device communication to the system.

```
import {setupBrokerContainer} from "../app-setup/container.setup";
import {WorkerMessage, Command, WorkerMessageData} from "./messaging.model";
import {IBrokerService} from "./broker.service";
import {Logger} from "bunyan";
import {ITaskService} from "./task.service";
import {IVariableTaskProvider, VariableTaskProvider} from
"../variable/variable";
import {IStorageSetup} from "../storage/storage";
import * as http from "http";
import * as io from "socket.io";
import {ContainerItem} from "../shared/IoC/ioc";
import {Config} from "../config";
import Socket = SocketIO.Socket;

const container = setupBrokerContainer();

export class MessagingProcess {

    static $inject = ["IBrokerService", "Logger", "ITaskService",
"IStorageSetup", "Config"];

    constructor(private brokerService: IBrokerService,
                private logger: Logger,
                private taskService: ITaskService,
                private storageSetup: IStorageSetup,
                private config: Config) {

        logger.info("Worker Process Starting...");

        this.handleRabbitInit();
        this.handleSocketServer();
        this.handleBrokerCleanUp();


        container.create<IVariableTaskProvider>(VariableTaskProvider);
    }

    private handleRabbitInit(): void {
        const MESSAGE = "message";
        process.on(MESSAGE, (message: WorkerMessage) => {
            switch (message.command) {
                case Command.subscribe:
                {
                    this.brokerService.subscribe(
                        message.payload.sensor,
                        message.payload.topic,
                        this.taskService.getTask(message.payload.taskKey),
                        message.payload.isRemovableHandler)
                        .then((handlerId: string) => {
                            process.send({
                                command: Command.handlerRegistered,
                                payload: {
                                    handlerId: handlerId
                                } as WorkerMessageData
                            } as WorkerMessage);
                        });
                        break;
                    }
                    case Command.registerSensor:
                    {

this.brokerService.registerSensor(message.payload.sensor);
                        break;
                    }
                    case Command.unsubscribe:
                    {

this.brokerService.unsubscribe(message.payload.handlerId);
                        break;
                    }
                    case Command.publish:
                    {
                        this.brokerService.publish(
                            message.payload.sensor,
                            message.payload.topic,
                            message.payload.data);
                        break;
                    }
                    case Command.registerJob:
                    {
                        this.brokerService.registerJob(
                            message.payload.sensor,
                            message.payload.topic);
                        break;
                    }
                }
        });

        this.logger.info("Sending broker process init message");

        process.send({
            command: Command.init
        });
    }

    private handleSocketServer(): void {
        this.storageSetup
            .initConnection()
            .then(() => {
                this.logger.info("Broker process started");
            });

        this.logger.info("Starting Socket server...");

        const server = http.createServer();

        const socketPort = parseInt(process.env.PORT || this.config.port,
undefined) + 1;

        server.listen(socketPort, () => {
            this.logger.info(`socket server listening on port:
${socketPort}`);
```

```
        });

        const socketServer = io.listen(server);

        socketServer.on("connection", (socket: Socket) => {
            this.logger.info("client connected");

            socket.on("disconnect", () => {
                this.logger.info("client disconnected");
            });
        });

        ContainerItem.create(container)
            .asNonInstantiatable()
            .forKey("SocketServer")
            .use(socketServer)
            .andRegister();
    }

    private handleBrokerCleanUp(): void {
        setTimeout(() => {
            this.brokerService.cleanUp();
            this.handleBrokerCleanUp();
        }, this.config.brokerCleanUpDelay);
    }
}

container.create<MessagingProcess>(MessagingProcess);
```

REST API and CRUD Operations (Sensor)

The snippets below are the classes that make up the full stack (from API Router to CRUD service) classes that enable Sensor management (Similar classes exist for Variables, Jobs, Streams, etc.).

```
import {CampModel} from "../shared/model/common.model";
import {NotEmpty} from "validator.ts/decorator/Validation";

export class Sensor extends CampModel {

    @NotEmpty()
    name: string;

    variables: string[];

    jobs: string[];

}
import {ISensorService} from "./sensor.service";
import {IVariableProvider} from "../variable/variable";
import {Sensor} from "./sensor.model";
import {IUserService, User} from "../user/user";
import {IJobProvider} from "../job/job";

export interface ISensorProvider {
    createSensor(sensor: Sensor, userId: string): Promise<Sensor>;
    updateSensor(sensor: Sensor): Promise<void>;
    deleteSensor(userId: string, sensorId: string): Promise<void>;
    getAllSensors(userId: string): Promise<Sensor[]>;
    getSensor(sensorId: string): Promise<Sensor>;
    validateSensor(sensor: Sensor): Promise<Sensor>;
}

export class SensorProvider implements ISensorProvider {

    static $inject = ["ISensorService", "IUserService", "IVariableProvider",
"IJobProvider"];

    constructor(private sensorService: ISensorService,
                private userService: IUserService,
                private variableProvider: IVariableProvider,
                private jobProvider: IJobProvider) {
    }

    createSensor(sensor: Sensor, userId: string): Promise<Sensor> {
        return this.sensorService
            .createSensor(sensor, userId)
            .then((sensor: Sensor) => {
                return this.userService
                    .addSensorToUser(userId, sensor._id)
                    .then(() => {
                        return sensor;
                    });
            });
    }
    updateSensor(sensor: Sensor): Promise<void> {
        return this.sensorService.updateSensor(sensor);
    }

    deleteSensor(userId: string, sensorId: string): Promise<void> {
        return this.sensorService
            .getSensor(sensorId)
            .then((sensor: Sensor) => {
                this.sensorService.deleteSensor(userId, sensorId);
                return sensor;
            })
            .then((sensor: Sensor) => {
                return this.userService
                    .removeSensorFromUser(userId, sensorId)
                    .then(() => {
                        return sensor;
                    });
            })
            .then((sensor: Sensor) => {
                this.variableProvider.deleteVariables(sensor.variables);
                this.jobProvider.deleteJobs(sensor.jobs);
            });
    }

    getAllSensors(userId: string): Promise<Sensor[]> {
        return this.userService
            .findUserById(userId)
            .then((user: User) => {
                return this.sensorService.getAllSensors(user.sensors);
            });
    }

    getSensor(sensorId: string): Promise<Sensor> {
        return this.sensorService.getSensor(sensorId);
    }

    validateSensor(sensor: Sensor): Promise<Sensor> {
        return this.sensorService.validateSensor(sensor);
    }
}
import {RouteController, DependableRequest} from
"../shared/service/router.service";
import {Request, Response, NextFunction} from "express";
import {Sensor} from "./sensor.model";
import {IContextService} from "../shared/service/context.service";
import {HttpStatusCodes} from "../http/http-status-codes";
import {ISensorProvider} from "./sensor.provider";

export class SensorRouter extends
RouteController<SensorRouterDependencyProvider> {

    static $inject = ["ISensorProvider", "IContextService"];

    constructor() {
        super(SensorRouterDependencyProvider);

        this.createPostRoutes();
        this.createGetRoutes();
        this.createDeleteRoutes();
    }
    private createPostRoutes(): void {
        this.router.post("/sensor",
            async(req: Request, res: Response, next: NextFunction) => {
                let request = req as
DependableRequest<SensorRouterDependencyProvider>;

                await
request.dependencies.sensorProvider.validateSensor(req.body)
                    .then((sensor: Sensor) => {
                        (<any>req).sensor = sensor;
                        next();
                    })
                    .catch((errs: Error[]) => {
                        res.status(HttpStatusCodes.BAD_REQUEST).json(errs);
                    });
            },
            async(req: Request, res: Response) => {
                let request = req as
DependableRequest<SensorRouterDependencyProvider>;

                await request.dependencies.sensorProvider
                    .createSensor((<any>req).sensor, (req as any).user._id)
                    .then((sensor: Sensor) => {
                        res.status(HttpStatusCodes.OK).json(sensor);
                    })
                    .catch((err: Error) => {

res.status(HttpStatusCodes.INTERNAL_SERVER_ERROR).json(err);
                    });
            });
    }

    protected routePrefix(): string {
        return "/sensor";
    }

    private createGetRoutes(): void {
        this.router.get("/sensor", async(req: Request, res: Response) => {
            let request = req as
DependableRequest<SensorRouterDependencyProvider>;

            await request.dependencies.sensorProvider
                .getAllSensors((req as any).user._id)
                .then((sensors: Sensor[]) => {
                    res.status(HttpStatusCodes.OK).json(sensors);
                })
                .catch((err: Error) => {

res.status(HttpStatusCodes.INTERNAL_SERVER_ERROR).json(err);
                });
        });

        this.router.param("id", async(req: Request, res: Response, next:
NextFunction, value: string) => {
            (<any>req).sensorId = value;
            next();
        });

        this.router.get("/sensor/:id", async(req: Request, res: Response) =>
{
            let request = req as
DependableRequest<SensorRouterDependencyProvider>;

            await request.dependencies.sensorProvider
                .getSensor((<any>req).sensorId)
                .then((sensor: Sensor) => {
                    res.status(HttpStatusCodes.OK).json(sensor);
                })
                .catch((err: Error) => {

res.status(HttpStatusCodes.INTERNAL_SERVER_ERROR).json(err);
                });

        });
    }

    private createDeleteRoutes(): void {
        this.router.delete("/sensor", async(req: Request, res: Response) =>
{
            let request = req as
DependableRequest<SensorRouterDependencyProvider>;

            await request.dependencies.sensorProvider
                .deleteSensor((req as any).user._id, req.query.id)
                .then(() => {
                    res.sendStatus(HttpStatusCodes.OK);
                })
                .catch((err: Error) => {

res.status(HttpStatusCodes.INTERNAL_SERVER_ERROR).json(err);
                });
        });
    }
}

class SensorRouterDependencyProvider {
    static $inject = ["ISensorProvider", "IContextService"];

    constructor(public sensorProvider: ISensorProvider,
                public contextService: IContextService) {
    }
}
import {Sensor} from "./sensor.model";
import {Logger} from "bunyan";
import {ISensorModelRepository} from "../storage/storage.service";
import {IValidator} from "../shared/service/validator.service";
import {IMessagingService} from "../messaging/messaging.service";

export interface ISensorService {
    createSensor(sensor: Sensor, userId: string): Promise<Sensor>;
    updateSensor(sensor: Sensor): Promise<void>;
    deleteSensor(userId: string, sensorId: string): Promise<void>;
    getAllSensors(sensorIds: string[]): Promise<Sensor[]>;
    getSensor(sensorId: string): Promise<Sensor>;
    validateSensor(sensor: Sensor): Promise<Sensor>;
    addVariableToSensor(id: string, variableId: string): Promise<Sensor>;
    removeVariableFromSensor(sensorId: string, variableId: string):
Promise<Sensor>;
    addJobToSensor(sensorId: string, jobId: string): Promise<Sensor>;
    removeJobFromSensor(sensorId: string, jobId: string): Promise<Sensor>;
}
```

```
export class SensorService implements ISensorService {

    static $inject = [
        "ISensorModelRepository",
        "IValidator",
        "Logger",
        "IMessagingService"
    ];

    constructor(private sensorRepository: ISensorModelRepository,
                private validator: IValidator,
                private logger: Logger,
                private messagingService: IMessagingService) {
    }

    createSensor(sensor: Sensor, userId: string): Promise<Sensor> {
        return this.sensorRepository.save(sensor)
            .then((sensor: Sensor) => {
                return Promise.all<void, Sensor>([
                    this.registerSensorWithBroker(sensor),
                    sensor
                ]);
            })
            .then((values: any[]) => {
                return <Sensor>values[1];
            })
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    getAllSensors(sensorIds: string[]): Promise<Sensor[]> {
        return this.sensorRepository
            .findMany(sensorIds)
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    getSensor(sensorId: string): Promise<Sensor> {
        return this.sensorRepository.find(sensorId)
            .then((sensor: Sensor) => {
                return sensor;
            })
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    updateSensor(sensor: Sensor): Promise<void> {
        return undefined;
    }

    deleteSensor(userId: string, sensorId: string): Promise<void> {
        return this.sensorRepository
            .delete(sensorId)
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    validateSensor(sensor: Sensor): Promise<Sensor> {
        let validate = (resolve: Function, reject: Function) => {
            let errors = this.validator.validate(sensor);
            if (errors && errors.length > 0) {
                reject(errors);
                return;
            }

            resolve(sensor);
        };

        return new Promise<Sensor>(validate);
    }

    addVariableToSensor(id: string, variableId: string): Promise<Sensor> {
        return this.sensorRepository.addVariable(id, [variableId])
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    removeVariableFromSensor(sensorId: string, variableId: string):
Promise<Sensor> {
        return this.sensorRepository
            .deleteVariable(sensorId, variableId)
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    addJobToSensor(sensorId: string, jobId: string): Promise<Sensor> {
        return this.sensorRepository.addJob(sensorId, [jobId])
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    removeJobFromSensor(sensorId: string, jobId: string): Promise<Sensor> {
        return this.sensorRepository
            .deleteJob(sensorId, jobId)
            .catch((err: Error) => {
                this.logger.error(err);
                return Promise.reject(err);
            });
    }

    private registerSensorWithBroker(sensor: Sensor): Promise<void> {
        return this.messagingService.registerSensor(sensor._id);
    }
}
```

Sensor Code

```
var SENSORID = "";
var TOGGLELED = "";
var LEDSTATE = "";
var LEDSTREAM = "";

var rabbit = require("rabbot");
var Gpio = require("onoff").Gpio;
var led = new Gpio(26, "out");
var express = require("express");
var app = express();

function handleMessage() {

    streamer();

    var handler = rabbit.handle(TOGGLELED, function (msg) {
        try {
            var parsedBody = msg.body;
            if (parsedBody.onOff == "1") {
                led.writeSync(1);
            } else {
                led.writeSync(0);
            }

            var ledState = led.readSync();

            console.log(ledState);

            publishState(LEDSTATE, ledState, false);
            msg.ack();
        }
        catch (err) {
            console.log(err);
            message.nack();
        }
    });
    console.log("waiting for message from publisher");
}

function publishState(variableId, onOff, numeric) {
    var p = rabbit.publish(SENSORID, {
        type: variableId,
        body: {
            variableId: variableId,
            value: convert(onOff, numeric)
        },
        routingKey: variableId
    });
}

function convert(onOff, numeric) {
    if (numeric) return onOff;

    return onOff === 1 ? true : false;
}

function streamer() {
    setTimeout(function () {
        var ledState = led.readSync();
        publishState(LEDSTREAM, ledState, false);

        streamer();
    }, 10 * 1000);
}

var bodyParser = require("body-parser");

app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(bodyParser.urlencoded({ extended: true }));

app.get("/", function (req, res) {
    if (TOGGLELED === "") {
        res.sendFile("SensorWebPage.html", { root: __dirname });
    } else {
        res.send("LED Sensor Activated.");
    }
});

app.post("/start", function (req, res) {
    TOGGLELED = req.body.toggleLedId;
    LEDSTATE = req.body.ledStateId;
    LEDSTREAM = req.body.ledStreamId;
    SENSORID = req.body.sensorId;

    rabbit.configure({
        connection: {
            uri: "amqp://iqcujvfc:tAoosDI1yhbRh1-
u4RK6Zb1KcaESx8E2@weasel.rmq.cloudamqp.com/iqcujvfc"
        },
        queues: [
            { name: TOGGLELED, subscribe: true, durable: true },
            { name: LEDSTATE, subscribe: true, durable: true }
        ],
        exchanges: [{ name: SENSORID, type: "topic", persistent: true,
durable: true }],
        bindings: [
            { exchange: SENSORID, target: TOGGLELED, keys: [TOGGLELED] },
            { exchange: SENSORID, target: LEDSTATE, keys: [LEDSTATE] }
        ]
    })
    .then(handleMessage);

res.send("LED Sensor Activated.");
});

app.listen(8080, function () {
    console.log("Listening...");
});
```

```
var TOGGLELED = "";
var LEDSTATE = "";
var LEDSTREAM = "";

var mqtt = require('mqtt');
var Gpio = require("onoff").Gpio;
var led = new Gpio(26, "out");

var express = require("express");
var app = express();

var client =
mqtt.connect("mqtt://ihoygctf:u78pw3kJdxjf@m12.cloudmqtt.com:14461");

var subscribed = false;

function startStreaming() {
    client.on('connect', function () {
        if (!subscribed) {
            client.subscribe(TOGGLELED, function () {
                subscribed = true;

                client.on('message', function (topic, message, packet) {
                    console.log(topic + " " + message);

                    var parsedBody = JSON.parse(message);
                    if (parsedBody.onOff == "1") {
                        led.writeSync(1);
                    } else {
                        led.writeSync(0);
                    }

                    var ledState = led.readSync();

                    client.publish(LEDSTATE, JSON.stringify({
                        variableId: LEDSTATE,
                        value: ledState
                    }), function () {
                        console.log("Message is published");
                    });
                });
            });

            streamer();
        }
    });
}

function streamer() {
    setTimeout(function () {
        var ledState = led.readSync();

        client.publish(LEDSTREAM, JSON.stringify({
            variableId: LEDSTREAM,
            value: ledState ? true : false
        }), function () {
            console.log("Message is published");
            streamer();
        });
    }, 10000);
}
```

```
var bodyParser = require("body-parser");

app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(bodyParser.urlencoded({ extended: true }));

app.get("/", function (req, res) {
    if (TOGGLELED === "") {
        res.sendFile("SensorWebPage.html", { root : __dirname});
    } else {
        res.send("LED Sensor Activated.");
    }
});

app.post("/start", function (req, res) {
    TOGGLELED = req.body.toggleLedId;
    LEDSTATE = req.body.ledStateId;
    LEDSTREAM = req.body.ledStreamId;

    startStreaming();

    res.send("LED Sensor Activated.");
});

app.listen(8081, function () {
    console.log("Listening...");
});
```

Appendix D: Dashboard URLs

The dash boards can be found at:

- **Amazon Setup**: http://camp-dashboard.azurewebsites.net
- **Azure Setup**: http://camp-dashboard-2.azurewebsites.net