University of Melbourne

A Framework for IoT Sensor Data Analytics and Visualisation in Cloud Computing Environments

Distributed Computing Project COMP90019

Kumaraswamy Krishnakumar (Student Id: 511942)
Ling wan (Student Id: 351451)

Project Supervisors:
Dr. Jayavardhana Gubbi
Prof. Rajkumar Buyya

TABLE OF CONTENTS

ABSTRACT **2**

Keywords: 2

Terminology and acronyms: 2

CHAPTER 1: INTRODUCTION **4**

Internet of Things frameworks: 4

What are sensors: 5

Where are sensors used: 6

Why do we need sensors: 7

Wireless Sensor Networks: 7

Why did we decide to use the 'cloud' platform to develop and host our application: 8

Marriage of cloud computing and sensor networks: 9

Overview of the project 9

What is missing: 9

A typical SaaS delivered application life cycle and associated activities 10

CHAPTER 2: TECHNOLOGY REVIEW **11**

Why did we choose Windows Azure platform: 11

Azure storage: 13

Windows Azure Application Deployment 13

Why are we using Aneka: 13

Why ASP.NET MVC: 14

3rd party libraries for visualization: 14

Why C#: 15

CHAPTER 3: DESIGN **16**

Managed Extensibility Framework (MEF): 16

Managing session state: 17

CHAPTER 4: IMPLEMENTATION **27**

CHAPTER 5: SAMPLE APPLICATION SCREEN LAYOUTS **33**

CHAPTER 6: TESTING **37**

CHAPTER 7: CONCLUSIONS AND FUTURE WORK **38**

REFERENCES **40**

Abstract

The planet, natural systems, human systems and physical objects have always generated enormous amounts of data but we could not hear it, see it or capture it. But now we can because all of these things are instrumented and connected and so we have access to it.

Internet of Things (IoT):

It is estimated there are over a billion internet users and rapidly increasing. But there are more things on the internet than there are people on the internet. This is what we generally mean when we say internet of things. There are millions and millions of devices with sensors that are linked up together using networks that generate a sea of data. With the benefit of integrated information processing capacity, industrial products will take on smart capabilities. They may also take on electronic identities that can be queried remotely, or be equipped with sensors for detecting physical changes around them. Such developments will make the merely static objects of today dynamic ones - embedding intelligence in our environment and stimulating the creation of innovative products and new business opportunities. The Internet of Things will enable forms of collaboration and communication between people and things, and between things themselves, so far unknown and unimagined.
With continuing developments in miniaturization and declining costs, it is becoming not only technologically possible but also economically feasible to make everyday objects smarter, and to connect the world of people with the world of things. Building this new environment however, will pose a number of challenges. Technological standardization in most areas is still in its infancy, or remains fragmented. Not surprisingly, managing and fostering rapid technological innovation will be a challenge for governments and industry alike. But perhaps one of the most important challenges is convincing users to adopt emerging technologies like RFID. Concerns over privacy and data protection are widespread, particularly as sensors and smart tags can track a user’s movements, habits and preferences on a perpetual basis. Fears related to nanotechnology range from bio-medical hazards to robotic control. But whatever the concern, one thing remains clear: scientific and technological advances in these fields continue to move ahead at breakneck speed. It is only through awareness of such advances, and the challenges they present, that we can reap the future benefits of a fair, user-centric and global Internet of Things [1],[2],[3], [13].

In parallel to all this but somewhat recent is the huge growth in cloud platforms. This project is about allowing users to get information, knowledge and wisdom from sensor data by using the power of cloud computing and to achieve that in a scalable and economical way.
We develop a web application to be made available as a software as a service (SaaS) for sensor data analytics and visualisation. The web application is a ASP.NET MVC application hosted on Windows Azure platform.

Keywords: Internet of Things, Sensors, Sensor data, Cloud platforms, Windows Azure, Aneka, ASP.NET MVC, SaaS, Data analytics, Data visualisation, Storage.

Terminology and acronyms:

IoT: Internet of things
ISV: Independent software vendor
MEF: Managed extensibility framework

RFID: Radio frequency identification
SDP: Service delivery platform
SaaS: Software as a service
SSO : Single Sign-On

Chapter 1: Introduction

The Internet of Things (IoT) is a multidisciplinary domain that covers a large number of topics from purely technical issues (e.g., routing protocols, semantic queries), to a mix of technical and societal issues (security, privacy, usability), as well as social and business themes. IoT applications, both existing and potential, are equally diverse. If we are to summarize all of them into one, it is enabling the machine perception of the real world and seamless interactions with it. Environmental and personal health monitoring, monitoring and control of industrial processes including agriculture, smart spaces, and smart cities are just some of the examples of IoT applications.

Figure 1.1 Internet of Things (Source: CISCO Internet of Things)

Internet of Things frameworks:

Internet of Things frameworks might help support the interaction between "things" and allow for more complex structures like Distributed computing and the development of Distributed applications. Currently, Internet of Things frameworks seem to focus on real time data logging solutions like Pachube: offering some basis to work with many "things" and have them interact. Future developments might lead to specific Software development environments to create the software to work with the hardware used in the Internet of Things.

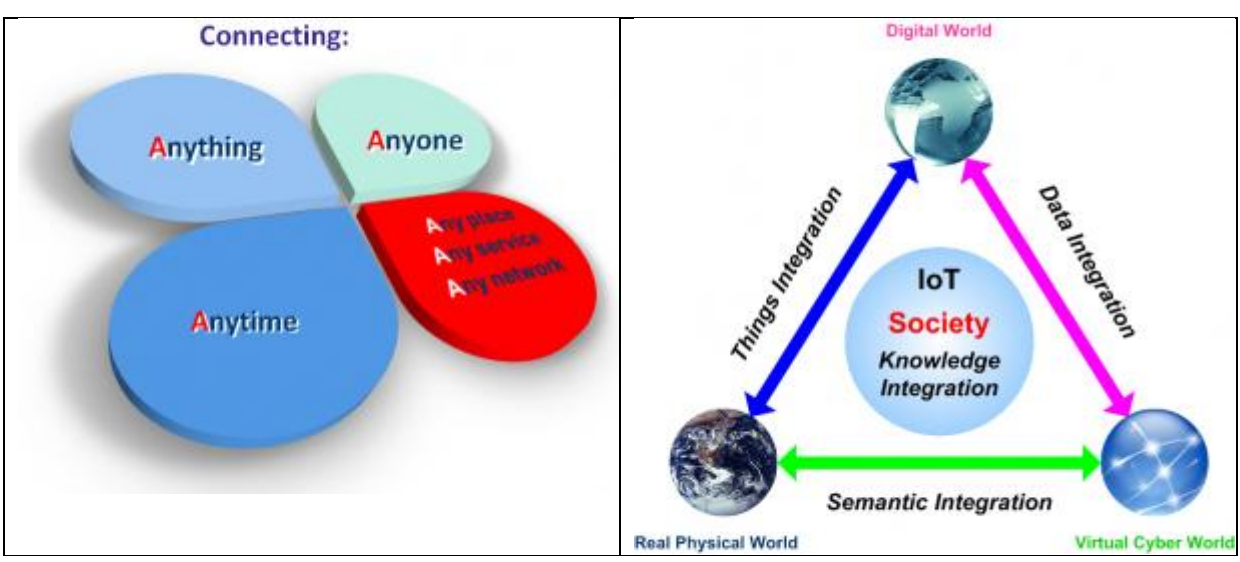


Figure 1.2 Internet of Things (Source: internet-of-things-research.eu)

What are sensors:

Sensors are devices that convert a physical parameter such as room temperature, blood pressure or wind speed into a signal that can be measured electrically or read by an observer. e.g. visual output from a glass thermometer. Sensors are a bridge between the physical world and the internet. They literally are the equivalent of human senses – sight, hearing, touch, smell and taste.

There are many types of sensors: chemical, magnetic, mechanical, position, pressure, temperature, CCD and CMOS image sensors, motion sensing, RFID etc.

One form of sensor technology is radio frequency identification (RFID), a method of identifying distinct items using radio waves. RFID is based on tags that contain microscopic chips used to store information about the item to which it is attached. The tag also contains a small, flat antenna. If the tag contains a power source, it is an active tag. If it depends on the reader for power, then it is a passive tag.

A passive tag read is activated by the radio waves emitted by a tag reader. When the antenna in the RFID tag encounters these radio waves, it forms a magnetic field that allows the tag to draw power and send information back to the reader. An active tag read occurs when the RFID tag has its own power source for the antenna and emits a signal that can be tracked.

Each year hundreds of millions of sensors are manufactured. The application of nanotechnology to sensors should allow improvements in functionality. In particular, new biosensor technology combined with micro and nanofabrication technology can deliver a huge range of applications. They should also lead to much decreased size, enabling the integration of 'nanosensors' into many other devices. We can also expect to see actuators that control movement on the nanoscale.

Sensor/actuator combinations will deliver 'smart' and precise functions in products and processes. For example, nanofabrication and inspection tools require sensors and actuator systems that can position objects with nanometre accuracy. In this way, sensors and actuators constitute another enabling technology. Many applications demand miniaturisation to reduce power consumption for integration into portable devices. Affordable mass production is also a prerequisite for sensors for consumer products, and for disposable devices such as sensors for medical diagnostics and pollution monitoring.

Sensors are becoming smarter, more accurate and cheaper. They will play an ever increasing role in just about every field imaginable.

(Source: drives.co.uk)

(Source: news.cnet.com)

(Source: spinwavesystems.com)

(Source: directindustry.com)

(Source: inertia.ece.virginia.edu)

(Source: ferret.com.au)

Figure 1.3 Wireless sensor nodes

Where are sensors used:

Sensors (wired and wireless) are ubiquitous and are in domestic appliances, medical equipment, industrial control systems, air-conditioning systems, aircraft, satellites, smoke detectors, robotics, missiles and toys. They are built into many consumer electronic devices, cars, medical devices, security and safety devices, and systems for monitoring pollution and environmental conditions. Sensors support applications across the economy - industrial processes, and those in construction, extractive industries, agriculture, health care and so on - and can be incorporated into new or existing products. In a city, ambient noise levels, CO2 levels, atmospheric temperature, humidity, wind speed, radiation levels etc. are monitored.

The following is a list of some application areas of wireless sensor networks:

- Asset and warehouse management
- Automotive
- Building monitoring and control
- Environmental monitoring
- Health care
- Industrial process control
- Military battlefield awareness
- Security and surveillance

Sensors can produce large volumes of continuous data over a period of time. The data can be live data, existing data, low resolution, high resolution etc [4],[7].

Why do we need sensors:

To get physical parameter data which are used to make decisions, control systems etc.
Once the physical parameter (temperature, displacement, acceleration, flow, chemical and biochemical parameters, such as concentrations of gases, ions or molecules, and molecular interactions) has been converted to an electrical equivalent it is easily input into a computer or microprocessor for manipulating, analysing and displaying.
Information from the data can then be used to make better decisions and smarter solutions leading to for example a smarter city which in turn results in better quality of life for the people.
To get information from these large volumes of data we need data storage, computational analysis and visualization infrastructure. To achieve this we have decided to develop a web application to be hosted in the cloud platform.

Wireless Sensor Networks:

A wireless sensor network (WSN) consists of spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants and to cooperatively pass their data through the network to a main location. The more modern networks are bi-directional, also enabling control of sensor activity. The development of wireless sensor networks was motivated by military applications such as battlefield surveillance; today such networks are used in many industrial and consumer applications, such as industrial process monitoring and control, machine health monitoring, and so on.
The WSN is built of "nodes" – from a few to several hundreds or even thousands, where each node is connected to one (or sometimes several) sensors. Each such sensor network node has typically several parts: a radio transceiver with an internal antenna or connection to an external antenna, a microcontroller, an electronic circuit for interfacing with the sensors and an energy source, usually a battery or an embedded form of energy harvesting. A sensor node might vary in size from that of a shoebox down to the size of a grain of dust, although functioning "motes"(demo video) of genuine microscopic dimensions have yet to be created. The cost of sensor nodes is similarly variable, ranging from a few to hundreds of dollars, depending on the complexity of the individual sensor nodes. Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed and communications bandwidth. The topology of the WSNs can vary from a simple star network to an advanced multi-hop wireless mesh network. The propagation technique between the hops of the network can be routing or flooding [5],[7].

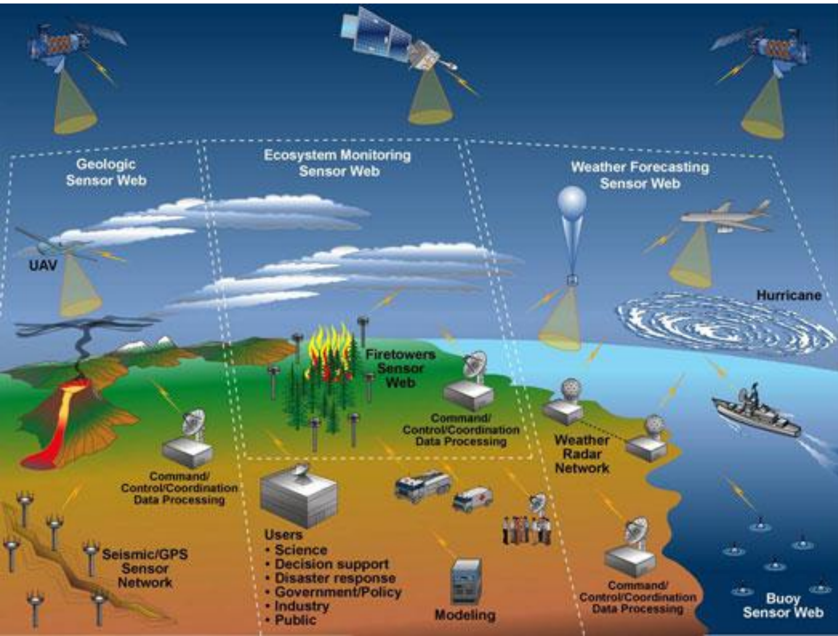


Figure 1.4 Wireless sensor networks (Source: CISCO WSN)

Why did we decide to use the 'cloud' platform to develop and host our application:

A platform is anything that can run applications and store data. In an organization's data centres, for example, you might have computers running Windows Server and other software that provide a platform for your in-house applications. A cloud platform is the same thing: It's a foundation for running applications and storing data. The biggest difference is that it runs in data centres owned by an external service provider, such as Microsoft, Amazon, Google, IBM, Rack Space etc., and it's accessed via the Internet.

Another difference is rather than rely on the exact same technology that your current in-house platform uses, cloud platforms often use something slightly different. For example, Microsoft's cloud platform runs Windows Azure rather than Windows Server.

Cloud platforms let even small organisations dream large!

So what are the benefits of using the cloud platform

- Faster deployment of new business capabilities
- Lower-risk business innovation
- Global scale and global reach
- More intelligent IT spending

What are the risks of using the cloud platform

- Outsourcing to an external provider
- Storing data outside your organization (This is clearly a risk but how much risk is appropriate or acceptable is a business decision.)
- Vendor lock-in (so far there is no easy way to port an application developed for one cloud provider to another cloud provider.)

[For pros and cons of Windows Azure platform please see page 10.]

Marriage of cloud computing and sensor networks:

Sensor networks promise to enable an entirely new class of distributed monitoring applications. For this promise to fully materialise tools from a variety of traditionally disparate disciplines have to be brought together, including signal processing, networking and protocols, database and information management and distributed algorithms.
Information processing in sensor networks is the central theme that binds all these components together and dictates how they must interoperate.
Internet has always played a key part of sensor networks. Cloud computing becomes extremely useful when the heavy computationally intensive work like data analytics needs to be done on sensor data. In this project we use the power of cloud computing to provide users the ability to do data analytics and visualisation on sensor data so that they can get useful information from sensor data.

Overview of the project

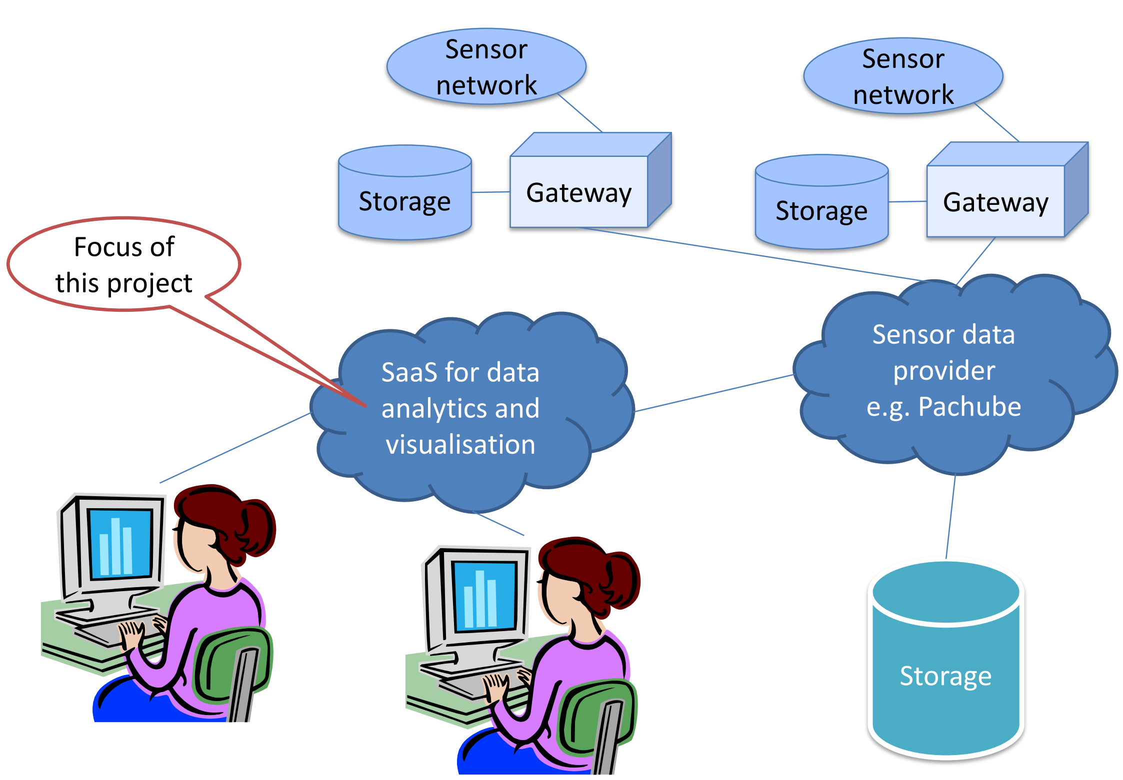


Figure: 1.5 Project overview

What is missing:

An application that users can use to do data analytics and data visualization including in 2D and 3D.

Aims of the project:

The major aim of this project is to create an infrastructure capable of handling large volumes of

sensor data generated by the internet of things and to provide related data analytics and visualization services on demand. One way of realising above objective is to provide a SaaS product (web application) that can be used to get information from sensor data, which can then be used to make decisions, take action etc. One of the key features of the software will be will be data analytics using various algorithms. These algorithms will be "hot-swappable". Another feature will be data visualization.

The software architecture should be flexible such that it can easily be extended in the future.

A typical SaaS delivered application life cycle and associated activities

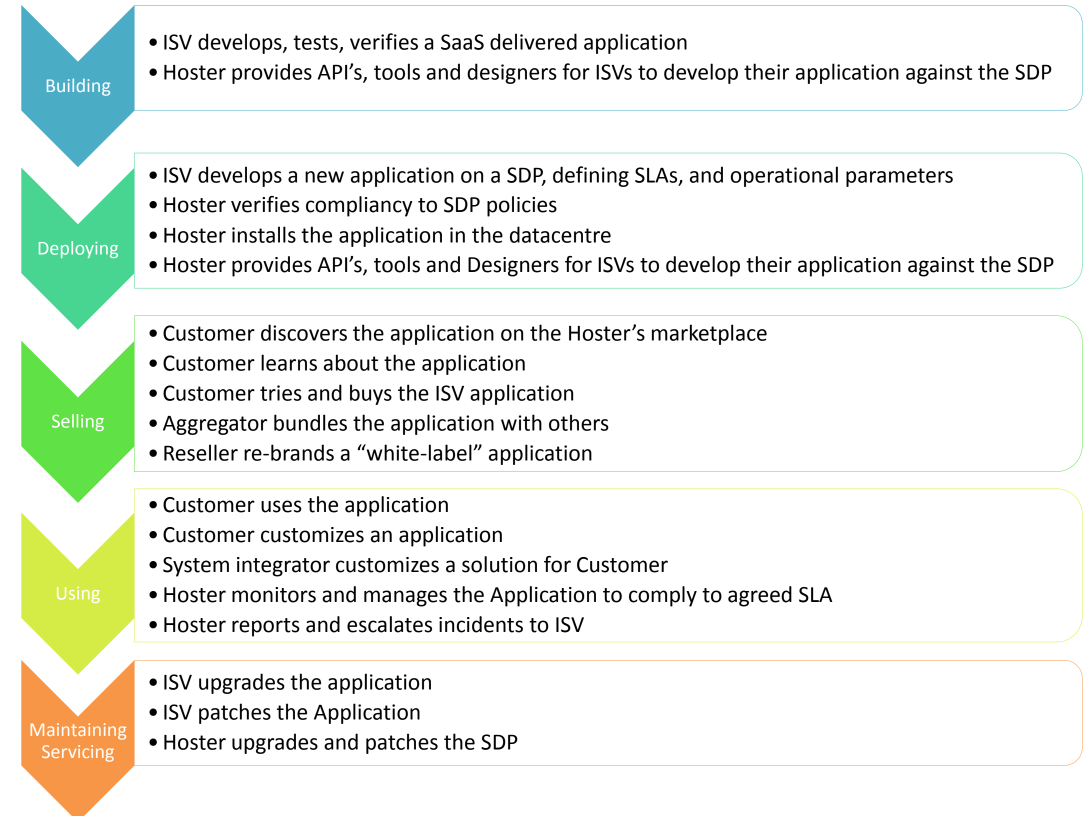


Figure 1.6 SaaS Application lifecycle

Chapter 2: Technology Review

We develop a single-tenant ASP.NET MVC (C#) application to support multiple customers on Windows Azure (PaaS) platform.
We develop the application incrementally and iteratively in an agile manner. The application will have a web role and a worker role. The application will be developed such that it can run correctly with more than one instance of each role.
The number of instances can be increased to handle bigger loads or decreased to save costs dynamically.
We use the standard proven layered architecture.
For this application we use three layers and they are the presentation layer, domain logic layer and the data access layer.
Domain logic will handle the data analytics. This work will be done in the worker role.
Data analytics is the computationally intensive part of the application. Here we use Aneka (please see page 12 for a description of Aneka) to distribute the work to multiple nodes. There are many more features in Aneka (e.g. combining private and public clouds, scheduling (costs vs time), QoS etc.) but for this project we are only using Aneka's task programming model to distribute the data analytics computation to multiple available nodes.
One of the key aims of this application is the ability to extend the application dynamically. i.e. without stopping the application.
To achieve this we need an extensible application architecture. There are a few ways to achieve this. We have decided to use the Managed Extensibility Framework (MEF) which is part of .Net Framework 4, to realise the extensible architecture.
Note: Once MEF is included in the application, many aspects (e.g. multi-visualization) of the application can be extended dynamically. For the purposes of this project, we will only be extending the data-analytics part.
MEF provides lot of functionality (load types, load assemblies, monitor folders, e.g. type catalogue, assembly catalogue, directory catalogue and aggregating catalogue, finds and loads all dependencies – the full object graph is taken care of) but we are only using the loading types dynamically part of it.
MEF also allows us to implement a key software principle: Open Close principle. i.e. Software entities should be open for extension, but closed for modification [8].

Application will be hosted in Windows Azure platform.

Why did we choose Windows Azure platform:

Millions of developers know how to create applications using the Windows Server programming model. Yet applications written for Windows Azure, don't exactly use this familiar model. While most of a Windows developer's skills still apply, Windows Azure provides its own programming model.
Why do we need another programming model? Why not just exactly replicate the familiar world of Windows Server in the cloud? Many vendors' cloud platforms do just this, providing virtual machines (VMs) that act like on-premises VMs. This approach, commonly called Infrastructure as a Service (IaaS), certainly has value, and it's the right choice for some applications.

Instead of IaaS, Windows Azure offers a higher-level abstraction that's typically categorized as Platform as a Service (PaaS). While it's similar in many ways to the on-premises Windows world, this abstraction has its own programming model meant to help developers build better applications. Applications built using the Windows Azure programming model can be easier to administer, more available, and more scalable (up or down) than those built on traditional Windows servers.

Administration:

In PaaS, the platform itself handles most of the administrative tasks. With Windows Azure, this means that the platform automatically takes care of things such as applying Windows patches and installing new versions of system software. The goal is to reduce the effort and the cost of administering the application environment.

Availability:

Whether planned or not, today's applications usually have down time for patches, application upgrades, hardware failures, and other reasons. With cloud platforms there is no need for any downtime. The Windows Azure programming model is designed to let applications be continuously available, even in the face of software upgrades and hardware failures.

Scalability:

The kinds of applications that people want to host in the cloud are often meant to handle lots of users. Yet the traditional Windows Server programming model wasn't explicitly designed to support Internet-scale applications. The Windows Azure programming model, however, was intended from the start to do this. Created for the cloud era, it's designed to let developers build the scalable applications that massive cloud data centres can support. Just as important, it also allows applications to scale down when necessary, letting them use just the resources they need and pay for only the computing resources used.

Windows Azure has three core components: Compute, Storage and Fabric. As the names suggest, Compute provides a computation environment with Web Role and Worker Role while Storage focuses on providing scalable storage (Blobs, Tables, Queue and Drives) for large-scale needs.

The Fabric Controller:

Windows Azure is designed to run in data centres containing lots of computers. Accordingly, every Windows Azure application runs on multiple machines simultaneously.

All the computers in a particular Windows Azure data centre are managed by an application called the fabric controller. The fabric controller is itself a distributed application that runs across multiple computers.

When a developer gives Windows Azure an application to run, he provides the code for the application's roles together with the service definition and service configuration files for this application. Among other things, this information tells the fabric controller how many instances of each role it should create. The fabric controller chooses a physical machine for each instance, then creates a VM on that machine and starts the instance running. The role instances for a single application are spread across different machines within this data centre.

Once it's created these instances, the fabric controller continues to monitor them. If an instance fails for any reason—hardware or software—the fabric controller will start a new instance for that role. While failures might cause an application's instance count to temporarily drop below what the

developer requested, the fabric controller will always start new instances as needed to maintain the target number for each of the application's roles.

Windows Azure programming model:

- A Windows Azure application is built from one or more roles.
- A Windows Azure application runs multiple instances of each role.
- A Windows Azure application behaves correctly when any role instance fails.

Azure storage:

Windows Azure offers blobs, tables, queues etc., as data storage options. They are a new type of data storage, they are fast and they are non-relational.
Storage must be external to role instances. This is to ensure if a role instance fails, any data it contains is not lost. So Windows Azure stores data persistently outside role instances. This way another role instance can now access data that otherwise would have been lost if that data had been stored locally on a failed instance.
Storage is replicated. Just as a Windows Azure application runs multiple role instances to allow for failures, Windows Azure storage provides multiple copies of data. Without this, a single failure would make data unavailable, something that's not acceptable for highly available applications.

Storage must be able to handle very large amounts of data. Traditional relational systems aren't necessarily the best choice for very large data sets. Since Windows Azure is designed in part for massively scalable applications, it must provide storage mechanisms for handling data at this scale.
We can use blobs for storing binary data and tables for storing large structured data sets.
If we need to use a traditional relational database then we can use SQL Azure [8],[9],[11].

Windows Azure Application Deployment

When we deploy the application, you can select the subregion (which at the moment determines the data centre) where you want to host the application. You can also define affinity groups that you can use to group inter-dependent Azure applications and storage accounts together in order to improve performance and reduce costs. Performance improves because Windows Azure co-locates members of the affinity group in the same data centre. This reduces costs because data transfers within the same data centre do not incur bandwidth charges. Affinity groups offer a small advantage over simply selecting the same subregion for your hosted services, because Windows Azure makes a "best effort" to optimise the location of those services.

Why are we using Aneka:

What is Aneka:

Aneka is a market oriented Cloud development and management platform with rapid application development and workload distribution capabilities developed by Manjrasoft. It is market oriented since it allows you to build, schedule, provision and monitor results using pricing, accounting, QoS/SLA services in private and/or public network environments.
Aneka is a platform independent, enterprise grid and cloud computing middleware for building and deploying parallel data and compute intensive applications. Aneka is both a development and execution environment supporting multiple programming models.
Aneka provides APIs and tools that enable applications to be virtualized over a heterogeneous

network.
Supported APIs include:

- Task Model for batch and legacy applications
- Thread Model for applications that use object oriented thread
- MapReduce Model for data intensive applications like data mining or analytics.
- Others such as MPI (Message Passing) and Actors (Distributive Active Objects/Agents) can be customized

How does Aneka help in our application:

(There are many other features of Aneka that can be used to develop a far more complete, powerful economical market oriented application. Please see the future work section at the end of the document.)
The most compute intensive part of the application is data analytics. It is in this data analytics part we will be using Aneka and in particular Aneka task programming model.
Task programming model provides the ability of expressing applications as a collection of independent tasks. Each task can perform different operations, or the same operation on different data, and can be executed in any order by the runtime environment. This is a scenario in which many scientific applications fit in and a very popular model for Grid Computing. Not surprisingly data analytics fits into this category and we use Aneka task programming model to execute the data analytics algorithms 12],[14].

Why ASP.NET MVC:

The MVC pattern lends itself naturally to separation of concerns. i.e. lends itself naturally to layered architecture. One advantage of this is we can develop various parts of the application independently. Another advantage is a loosely coupled application. It also makes it a lot easier to develop unit tests, which goes a long way in catching defects early. This in turn leads to better quality software in less time. Developing in ASP.NET MVC means 'close to the metal development' allowing us to produce lean and responsive html pages. There are many libraries available to make development a lot easier. E.g. MVC Futures library.
Urls are user friendly, is considered part of the U.I., and makes better user experience. Urls are also search engine friendly.

3rd party libraries for visualization:

Google Visualization API are powerful visualization techniques to bring new life and usefulness to large datasets. It is a pure javascript library that does not depend on any add-ons. e.g. flash plugin etc. It uses svg as its default graphics format but if svg is not supported by the browser it uses vml. Because svg (and vml) are vector based graphic formats the graphs never become pixelated regardless of screen resolutions or magnification. It enables you to add live charts, graphs, tables etc., to your web pages.
Google Visualization API, which is designed to make it easier for a wide audience to make use of advanced visualization technology, and do so in a way that makes it quick and easy to integrate with new visualizations.
There are a two key design motivation for Google Visualization API: simplicity and ubiquity.
There are also many C# and Java libraries that wrap the Google javascript api's thereby allowing it to be programmed in C#, Java etc.

There are other tools that can be used for visualization (e.g. Standford reserchers Protovis, IBMs Many eyes etc.) but because of the simplicity and power of Google Visualization API we decided to use it in this project.

We can display sensor data using 2D maps and graphs and 3D maps are used to show location of sensors in 3-dimension. For this we use latitude, longitude and altitude.
Using the Google visualisation api is as easy as:

Loading the Google maps api from the Google cdn network.
Prepare the sensor node information in the appropriate format – Google dataset.
Add earth as one of the map type: map.addMapType(G_SATELLITE_3D_MAP);
Create a marker.
Add a listener.

Why C#:

The name "C sharp" was inspired by musical notation where a sharp indicates that the written note should be made a semitone higher in pitch. Due to technical limitations of display (standard fonts, browsers, etc.) and the fact that the sharp symbol (U+266F ♯ music sharp sign (HTML: ♯)) is not present on the standard keyboard, the number sign (U+0023 # number sign (HTML: #)) was chosen to represent the sharp symbol in the written name of the programming language. However, when it is practical to do so (for example, in advertising or in box art), Microsoft uses the intended musical symbol.
C# was used as the programming language of choice as it is a modern object oriented language with many features implemented beautifully which makes it a pleasure developing with. C# is based on the 'C' family of languages making it accessible to large number of C, C++ and Java developers. Some of the features of C# are LINQ, lambda expressions, extension methods, anonymous methods, dynamic typing etc. The LINQ extensions and the functional imports help developers reduce the amount of "boilerplate" code that is included in common tasks like querying a database, parsing an xml file, or searching through a data structure, shifting the emphasis onto the actual program logic to help improve readability and maintainability.

Chapter 3: Design

Managed Extensibility Framework (MEF):

What is MEF:

The Managed Extensibility Framework is a composition layer for .NET that improves the flexibility, maintainability and testability of large applications. MEF can be used for third-party plugin extensibility, or it can bring the benefits of a loosely-coupled plugin-like architecture to regular applications.
It is a library for creating lightweight, extensible applications. It allows application developers to discover and use extensions with no configuration required. It also lets extension developers easily encapsulate code and avoid fragile hard dependencies. MEF not only allows extensions to be reused within applications, but across applications as well.
Using MEF is as easy as, export it, import it and compose it.

What problems does MEF solve:

MEF presents a simple solution for the runtime extensibility problem. Until now, any application that wanted to support a plugin model needed to create its own infrastructure from scratch. Those plugins would often be application-specific and could not be reused across multiple implementations.
MEF provides a standard way for the host application to expose itself and consume external extensions. Extensions, by their nature, can be reused amongst different applications. However, an extension could still be implemented in a way that is application-specific. Extensions themselves can depend on one another and MEF will make sure they are wired together in the correct order (another thing you won't have to worry about).
MEF offers a set of discovery approaches for your application to locate and load available extensions. MEF allows tagging extensions with additional metadata which facilitates rich querying and filtering.
One of the key design goals of our web application is, it should be extensible and this is the reason we decided to use MEF.
With MEF we can use different algorithms (as and when it becomes available) for sensor data analytics: e.g. drop an analytics assembly into a folder and it instantly becomes available to the application [10].

The problem of updating the analytics assembly:

We use a separate assembly (analytics load dll) from the main application to load the types from the analytics dll into a separate new app domain. i.e. the analytics_load.dll is loaded into a separate app domain. This analytics_load.dll then loads the required types using some MEF magic from the analytics dll.
Once the types are loaded we execute the required methods in the types and on completion, the new app domain is unloaded along with the dlls. The main reason for doing this is the analytics dll will not be locked once the new app domain is unloaded and so can be updated as required. Once updated will be immediately available to users.
We had considered copying the analytics dll into the bin folder of the application and since dll's in the bin folder are not locked (they are shadow copied and used) they can be updated any time.

Unfortunately copying into the bin folder at runtime is not possible.
We also tried to copy the dll to a different folder and include that folder for shadow copying which means the dll will not be locked and so can be updated anytime. But the methods used to assign a folder for shadow copying has been deprecated and so it was decided not to use these methods.

Managing session state:

Considering more than one instance of the web role will be running we cannot use the traditional in-memory session state provider to store session data. In the past in a web farm scenario people have for example used the database to store session data. The same approach can be taken in the Azure platform by changing the in-memory session state provider to a TableStorage SessionState provider. So we can still use the Request.Session object to store session data but will be stored in the table storage so that all instances of the web role can access the same data.
Another approach to session state is to store the entire session state data on the client using javascript and AJAX.
The third approach is similar to ViewState in web forms, where the session state is serialised and deserialised and passed between pages.
ViewState in web forms was a contentious issue. But with the help of MVC Futures library serialising and deserialising session state can be done in an efficient and secure manner.

Comparing options for managing session state in the Azure platform:

	Table Storage Session State provider	Store it on the client using javascript and AJAX	Store it on the client similar to ViewState but by using the MVC Futures library
Simplicity	When session data is no longer required need to clear the tables	Complex; also difficult to unit test	Moderately complex
Cost	Moderate	Lowest	Higher cost due to higher bandwidth usage
Performance	Introduces some latency	Best	Some latency
Scalability	Good	Good	Good
Robustness	Moderate	Fragile	Best
User Experience	Good	Best	Good
Security	Good	Good	Moderate

Looking at the table above it looks like option three is a reasonable option.

As mentioned in the technology review section we have opted for the standard layered architecture

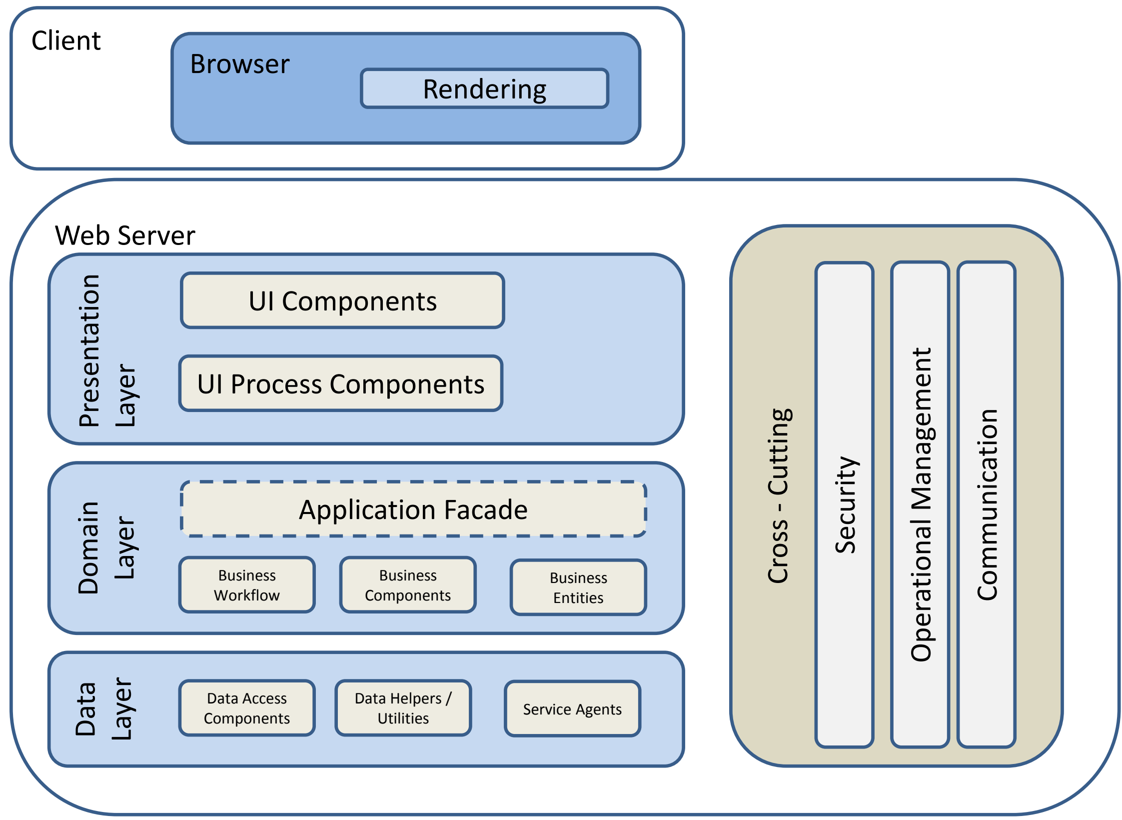


Figure 3.1 Layered architecture

The application context diagram:

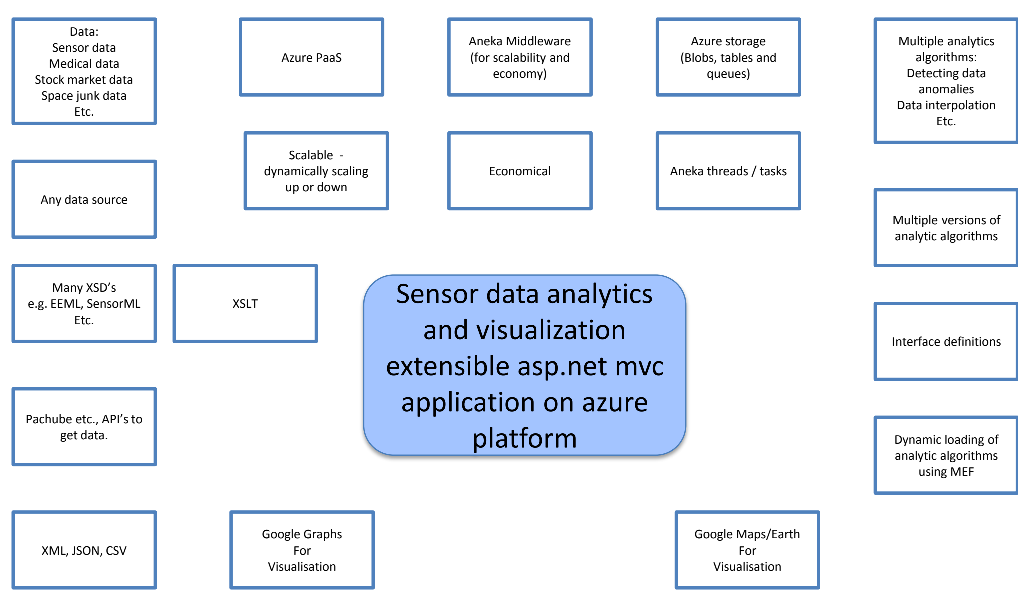


Figure 3.2 System context diagram

There are a number of design issues to consider:

Data Isolation and Sharing
Data Security
Data Storage and Extensibility
Identity Management
Multi-tenancy
On-premise or Off-premise
Performance
Service Composition
Service Integration
Service Management

Security issues in a typical web application:

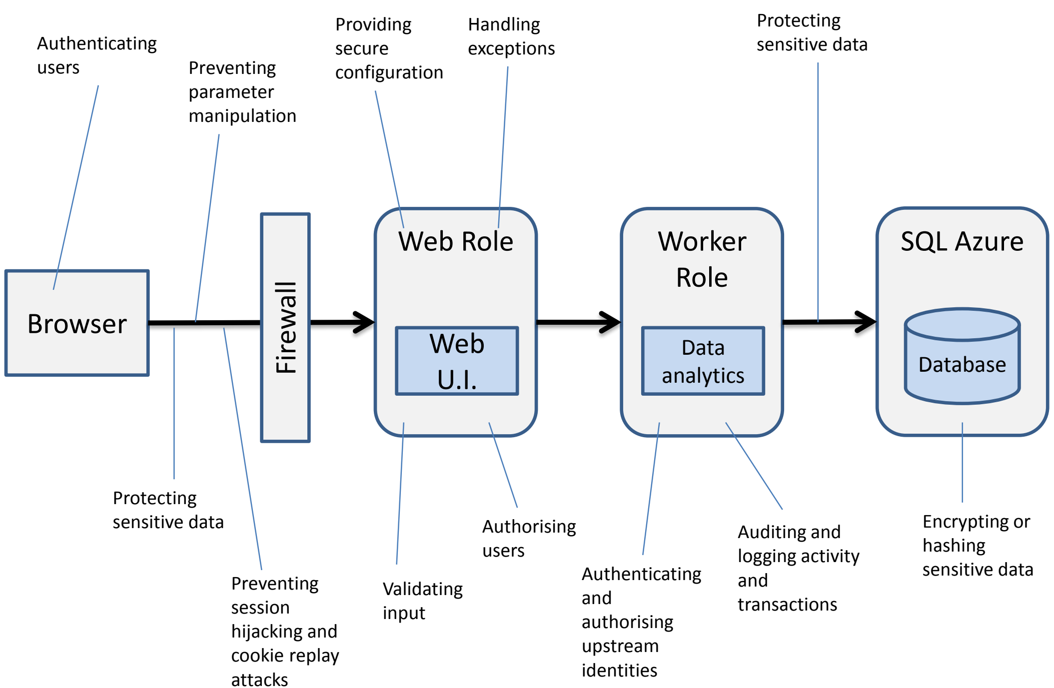


Figure 3.3 Security issues in Web applications

Identity Management

All applications and services must manage user identity. This is particularly important in cloud-based scenarios that can potentially serve a very large number of customers and each of these customers may have their own identity framework. The ideal solution is a solution that takes advantage of the customers existing on-premises or federated directory service to enable single sign on (SSO) across their local and all external hosted services. This reduces the development effort of building individual and separate identity management systems. SSO allows users to access the application or service using their existing credentials.

To enable such a scenario, we need to adopt a solution based on industry standards that interoperate across platforms and organisational boundaries. We can consider a claims-based identity model based on a federated identity service. This helps to decouple applications and services from the authentication mechanism.

Windows Azure - One or more instances of web roles and worker roles:

Every Windows Azure application consists of one or more roles. When it executes, an application that conforms to the Windows Azure programming model must run at least two copies—two distinct instances—of each role it contains. Each instance runs as its own VM.

Every instance of a particular role runs the exact same code. In fact, with most Windows Azure applications, each instance is just like all of the other instances of that role—they're interchangeable. For example, Windows Azure automatically load balances HTTP requests across an application's Web role instances. This load balancing doesn't support sticky sessions, so there's no way to direct all of a client's requests to the same Web role instance. Storing client-specific state, such as a shopping cart, in a particular Web role instance won't work, because Windows Azure provides no way to guarantee that all of a client's requests will be handled by that instance. Instead, this kind of state must be stored externally, for example in SQL Azure.
An application that follows the Windows Azure programming model must be built using roles, and it must run two or more instances of each of those roles. A Windows Azure application behaves correctly when any role instance fails. If all instances of a particular role fail, an application will stop behaving as it should—this can't be helped. The requirement to work correctly during partial failures is fundamental to the Windows Azure programming model. In fact, the service level agreement (SLA) for Windows Azure requires running at least two instances of each role. Applications that run only one instance of any role can't get the guarantees this SLA provides [11].

How load balancing happens with multiple instances:

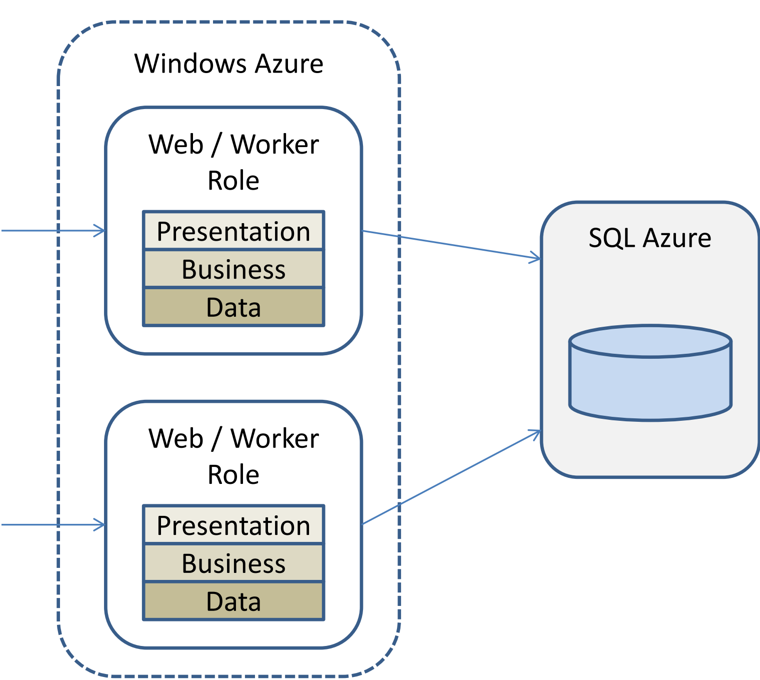


Figure 3.4 Load balancing in Azure platform

Sequence diagram of displaying selected feed sensor data:

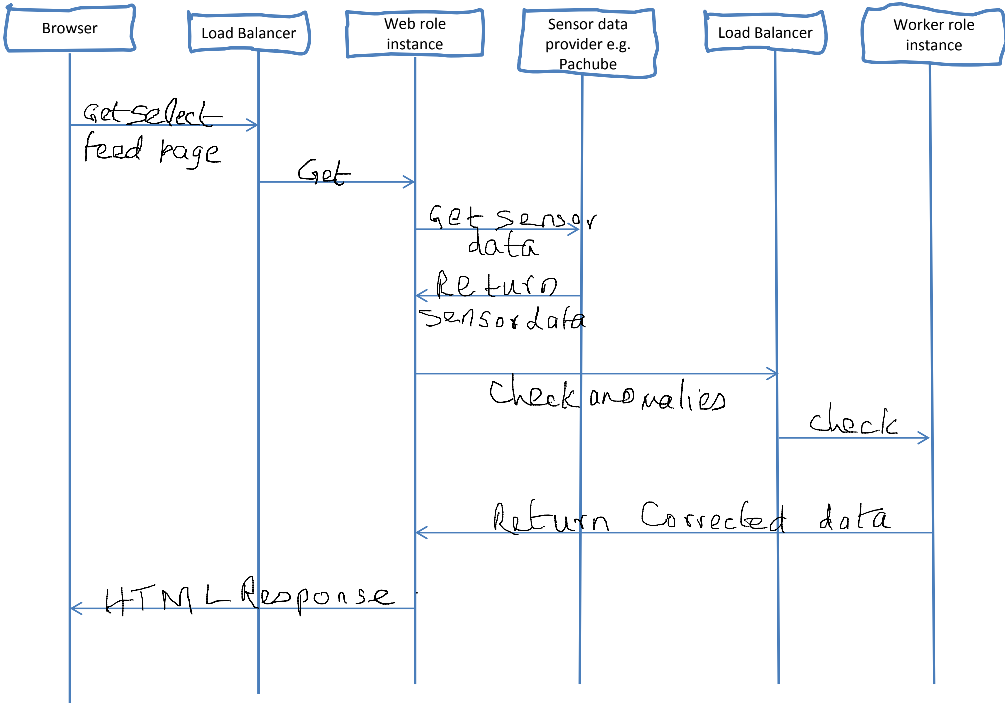


Figure 3.5 Displaying selected feed sensor data

Sequence diagram of the login process:

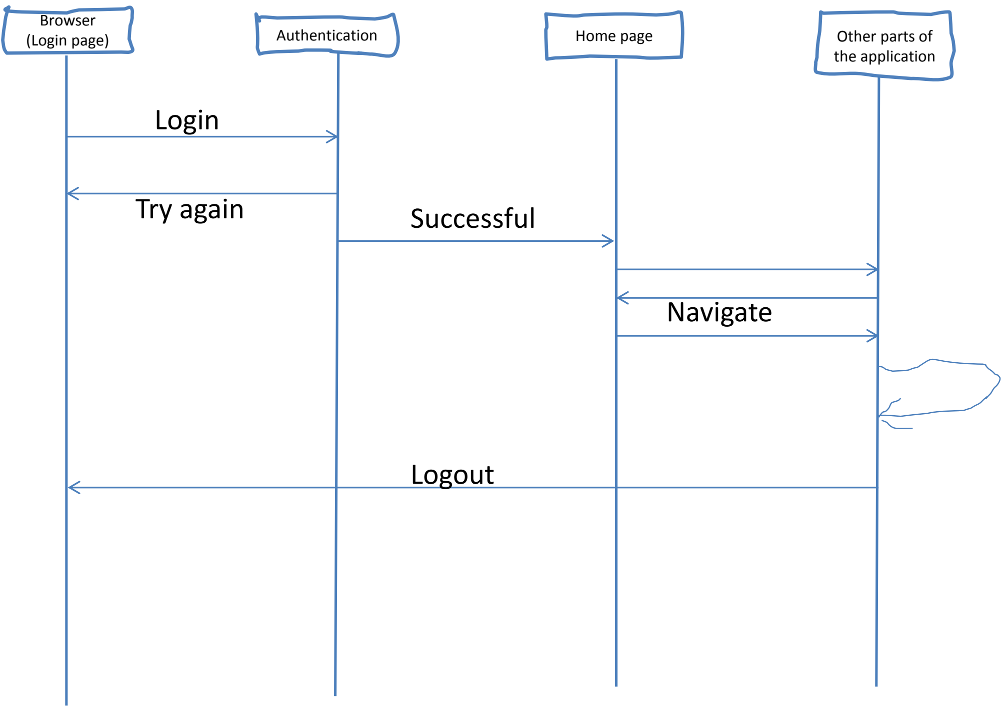


Figure 3.6 Login sequence diagram

Interface definitions used:

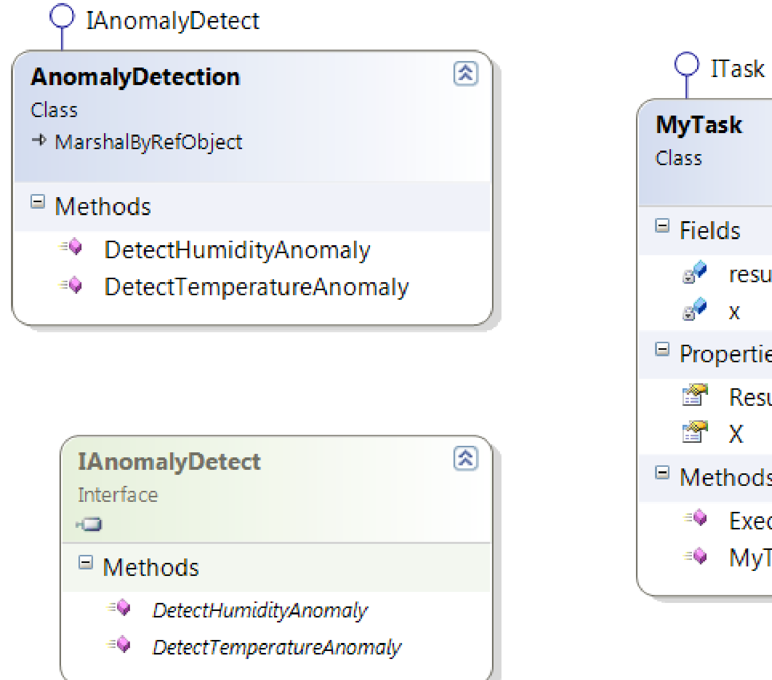


Figure 3.7 Interface definitions

Class diagram from the main application:

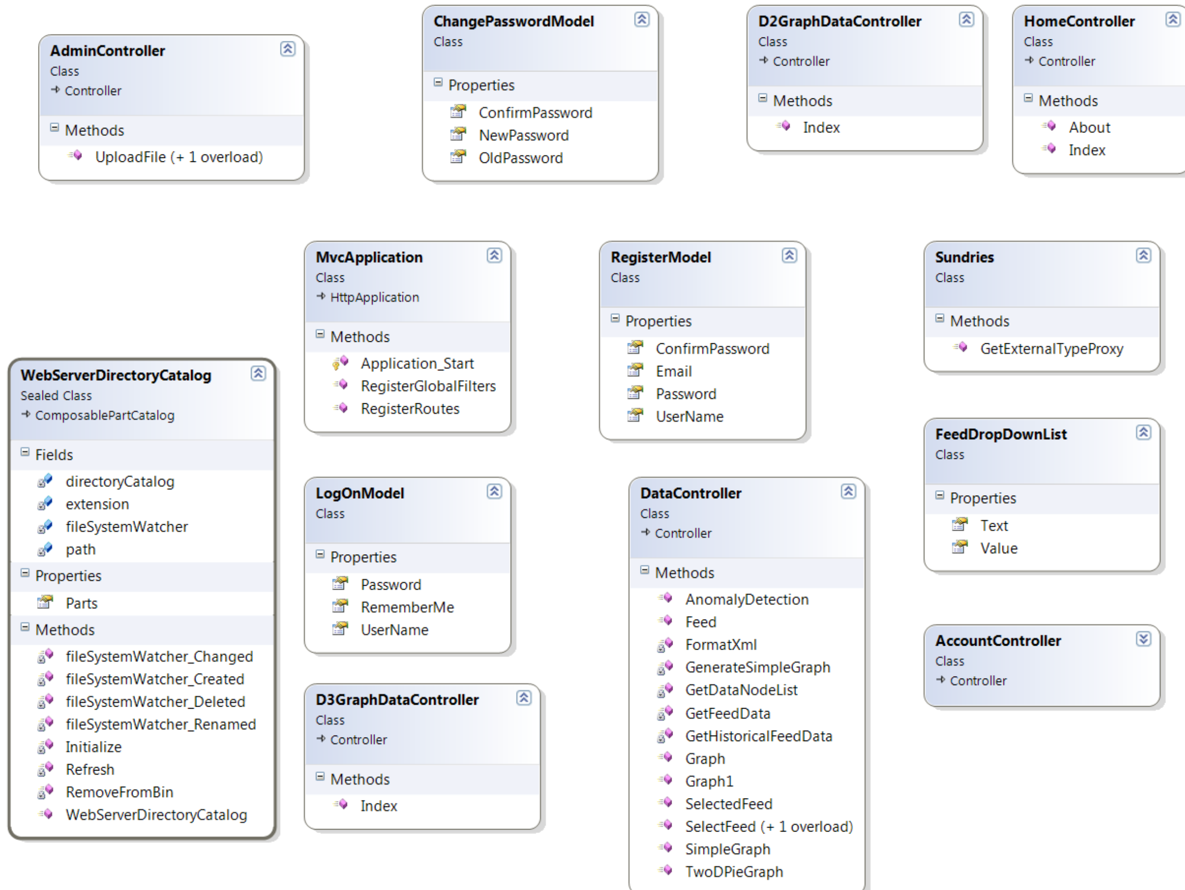


Figure 3.8 Class diagram

Chapter 4: Implementation

How we load analytics assembly at run time using MEF:

```
namespace AnalyticsLoad
{
    public class DataAnomalyDetection : MarshalByRefObject
    {

        //the type i am getting at runtime using MEF.
        [Import(typeof(IAnomalyDetect))]
        public IAnomalyDetect anomalyDetect;

        public DataAnomalyDetection()
        {
        }

        public string LoadExternalAssembly(string localStoragePath)
        {
            General gen = new General();
            gen.CopyAssembly(localStoragePath);

            ComposeTypeFromExternalAssembly(localStoragePath);

            return "Successfully executed.";
        }

        //compose external library. this is where the MEF magic happens!
        private void ComposeTypeFromExternalAssembly(string localStoragePath)
        {

            var catalog = new DirectoryCatalog(localStoragePath);

            var Container = new CompositionContainer(catalog);

            Container.ComposeParts(this);

            try
            {
                //Fill the imports of this object
                Container.ComposeParts(this);
                //Container.SatisfyImportsOnce(this);

            }
            catch (CompositionException compositionException)
            {
                //should do something
            }

        }

    }
}
```

How we load an assembly into a separate domain, use it and then unload it:

```
namespace SensorDataViz.ExtLibrary
{
    public class Sundries
    {

        public string GetExternalTypeProxy()
        {
            //it is in this method the majority of the work gets done by using
the 'external' dll.

            var localResource =
RoleEnvironment.GetLocalResource("extAssembliesStorage");
            string localStoragePath = localResource.RootPath;

            //Create new appDomain
            AppDomain newDomain = AppDomain.CreateDomain("NewAppDomain");

            // Determine /bin path
            string binPath =
Path.Combine(AppDomain.CurrentDomain.BaseDirectory, "bin");
            string assemblyPath = binPath + "\\AnalyticsLoad.dll";

            DataAnomalyDetection proxy =
(DataAnomalyDetection)newDomain.CreateInstanceFromAndUnwrap(assemblyPath,
"AnalyticsLoad.DataAnomalyDetection");

            //this copies the external assembly from azure storage to local
storage and loads types using MEF.
            string aa = proxy.LoadExternalAssembly(localStoragePath);

            string correctedData =
proxy.anomalyDetect.DetectTemperatureAnomaly("temp sample data");

            AppDomain.Unload(newDomain);

            return correctedData;

        }
    }
}
```

Getting the required feed from Pachube:

```
private string GetHistoricalFeedData(string feedNumber, string dataId)
        {
            //here we get the last 24 hours of data

            string curTimeStamp = DateTime.Now.AddDays(-1).ToString("yyyy-MM-
ddTHH:mm:ssZ");

            string myURI1 = @"http://api.pachube.com/v2/feeds/";
```

```
            string myURI = myURI1 + feedNumber + "/datastreams/" + dataId +
".xml?start=" + curTimeStamp + "&interval=24hours";
            string myKey = @"hABmde-aaaaaaaaa-o";

            var myWC = new WebClient();

            myWC.Headers.Add("X-PachubeApiKey", myKey);

            string tempData = myWC.DownloadString(myURI);

            return tempData;

        }
```

Displaying feed data in tabular format and Google map:

```
@using System.Xml;

@{
    ViewBag.Title = "Feed";
}

<h2>@ViewBag.Message</h2>

@{


XmlDocument doc = new XmlDocument();
doc.LoadXml(ViewBag.XmlFeed.ToString());


XmlNode titleNode = doc.GetElementsByTagName("title")[0];
XmlNode statusNode = doc.GetElementsByTagName("status")[0];
XmlNode websiteNode = doc.GetElementsByTagName("website")[0];
XmlNode emailNode = doc.GetElementsByTagName("email")[0];

XmlNode nameNode = doc.GetElementsByTagName("name")[0];
XmlNode latNode = doc.GetElementsByTagName("lat")[0];
XmlNode lonNode = doc.GetElementsByTagName("lon")[0];
XmlNode eleNode = doc.GetElementsByTagName("ele")[0];

}

Title: @titleNode.InnerText
<br />
Status:@statusNode.InnerText
<br />
Website:@websiteNode.InnerText
<br />
Email:@emailNode.InnerText
<br />
Name:@nameNode.InnerText
<br />
Latitude:@latNode.InnerText
<br />
Longitude:@lonNode.InnerText
```

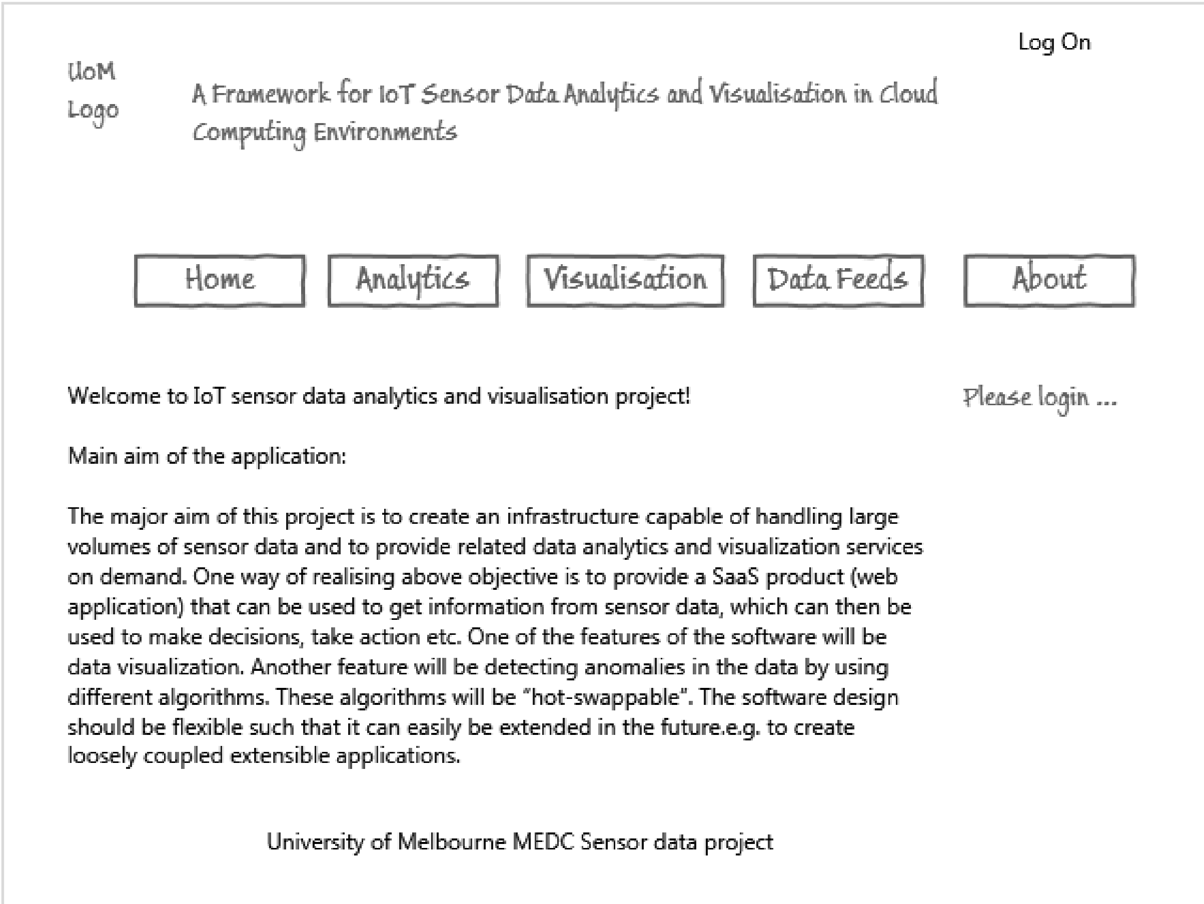
```
<br />
Elevation:@eleNode.InnerText
<br /><br />

@{
    XmlNodeList nodeList = ViewBag.DataNodeList as XmlNodeList;
}

<table border="0" cellpadding="0" cellspacing="0">

@foreach (XmlNode node in nodeList)
{

    foreach (XmlNode child in node.ChildNodes)
    {

    <tr>
        <td>
        @child.Name
        </td>
        <td>
        @child.InnerText
        </td>
    </tr>
    }

    <tr>
        <td> </td>
        <td> </td>
    </tr>

}
</table>

<br /><br />


<script
src="http://maps.google.com/maps?file=api&amp;v=2&amp;sensor=false&amp;key=ABQ
IAAAAaaaaaaaaaaaaaaaaaaaaaaaaaaaaO2ASw" type="text/javascript"></script>
    <script type="text/javascript">
        function loadXMLDoc(dname) {
            try //Internet Explorer
        {
                xmlDoc = new ActiveXObject("Microsoft.XMLDOM");
            }
            catch (e) {
                try //Firefox, Mozilla, Opera, etc.
             {
                    xmlDoc = document.implementation.createDocument("", "",
null);
                }
                catch (e) { alert(e.message) }
            }
            try {
                xmlDoc.async = false;
```

```
                xmlDoc.load(dname);
                return (xmlDoc);
            }
            catch (e) { alert(e.message) }
            return (null);
        }


        var map;
        var ge;

        google.load("maps", "2.x");

        function init() {

            lat = @latNode.InnerText

            lon = @lonNode.InnerText

            eld = @eleNode.InnerText

           tag = 
xmlDoc.getElementsByTagName("tag")[5].childNodes[0].nodeValue;
            max = 
xmlDoc.getElementsByTagName("max_value")[0].childNodes[0].nodeValue;
            min = 
xmlDoc.getElementsByTagName("min_value")[0].childNodes[0].nodeValue;

            map = new GMap2(document.getElementById('map3d'));
            map.setCenter(new GLatLng(lat, lon), 12);

            var mapui = map.getDefaultUI();
            mapui.maptypes.physical = false;
            map.setUI(mapui);

            // add 'Earth' as one of the map types
            map.addMapType(G_SATELLITE_3D_MAP);

            // create a marker
            var marker = new GMarker(new GLatLng(lat, lon));

            GEvent.addListener(marker, "click", function () {
                var html = '<div style="width: 210px; padding-right: 10px">' + 
tag + ': Maximum:' + max + '; Minimumin:' + min + '</div>';
                marker.openInfoWindowHtml(html);
            });

            map.addOverlay(marker);
            google.maps.Event.trigger(marker, "click");


            // do stuff with Earth when the user switches to it
            GEvent.addListener(map, 'maptypechanged', function () {
                if (ge)
                    return;
```

```
                map.getEarthInstance(function (pluginInstance) {
                    ge = pluginInstance;
                    doStuffWithEarth();
                });
            });

            //doStuffWithEarth();
        }

        function doStuffWithEarth() {
            document.getElementById('installed-plugin-version').innerHTML =
ge.getPluginVersion().toString();
        }

        google.setOnLoadCallback(init);

    </script>

    <div id="map3d" style="width: 500px; height: 380px;"></div>
    <br/>
    <div>Installed Plugin Version: <span id="installed-plugin-version"
style="font-weight: bold;">Loading...</span></div>
```

Chapter 5: Sample Application Screen Layouts

Home Screen: (Web browser interface)



Analytics Screen: (Web browser interface)

Log On

UoM
Logo

A Framework for IoT Sensor Data Analytics and Visualisation in Cloud Computing Environments

Home | Analytics | Visualisation | Data Feeds | About

Upload data file
Upload analytics assembly

Analytics

please select the required link

University of Melbourne MEDC Sensor data project

Visualisation Screen: (Web browser interface)

Log On

UoM
Logo

A Framework for IoT Sensor Data Analytics and Visualisation in Cloud Computing Environments

Home | Analytics | Visualisation | Data Feeds | About

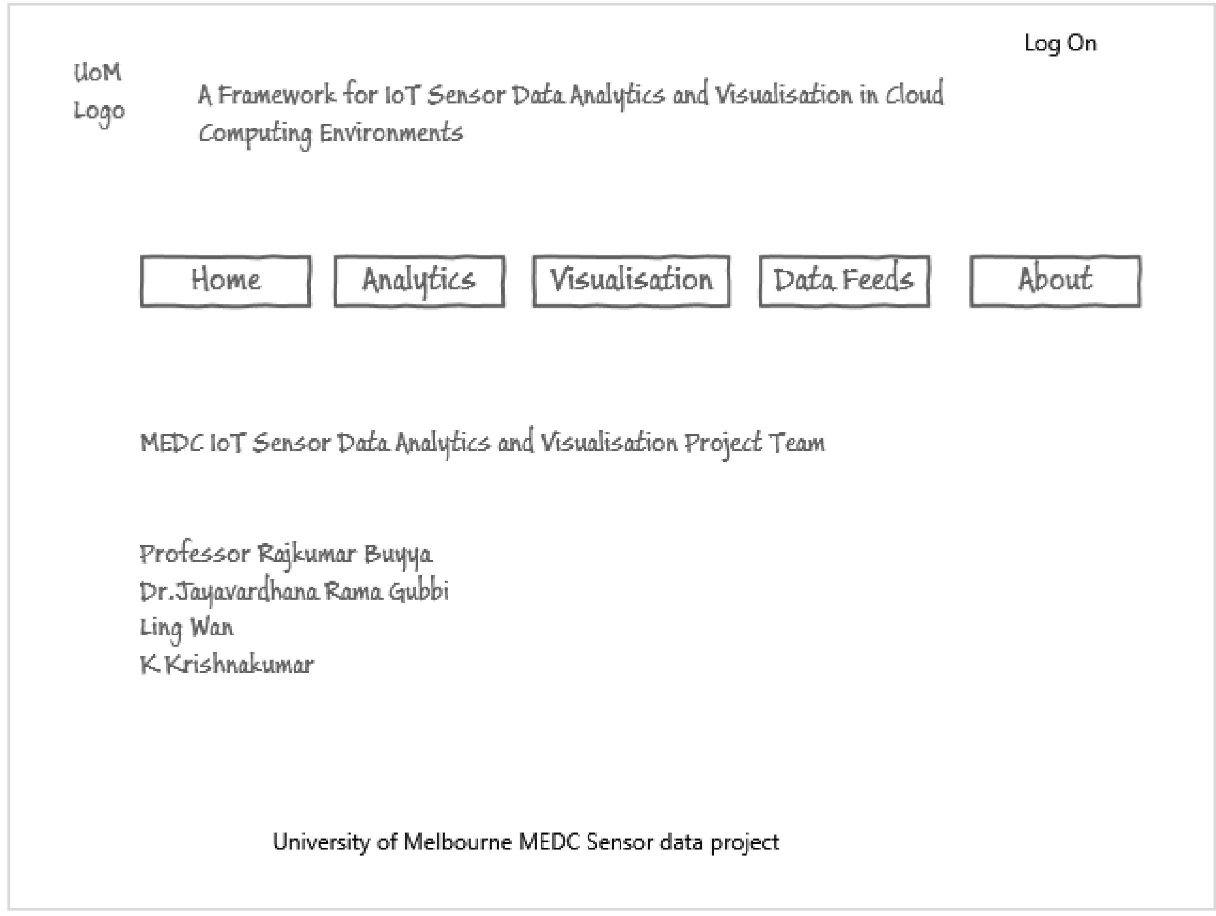
2D-Graph
3D-Graph

Visualisation

please select the required link

University of Melbourne MEDC Sensor data project

About Screen: (Web browser interface)



Chapter 6: Testing

From unit tests point of view, we must look at Azure applications from its components and their dependencies. You typically employ the following items to test your Azure applications:

- a unit testing framework to define and run the tests
- a mocking framework to help you isolate dependencies and build narrowly scoped unit tests
- tools that provide increased code coverage to aid automatic unit test generation

Unit testing.

Security: users will be authenticated before they can use the application.

Should be responsive: i.e. web pages should load quickly.

Should be available: Application should be always available even if running a bit slowly.

Should be scalable: Ability to handle large volumes of data and large number of simultaneous users.

Ability to specify different data feeds.

Ability to visualise data: e.g. Graphs, Maps (2D, 3D).

Ability to detect anomalies.

Chapter 7: Conclusions and future work

Our project was essentially a proof of concept application to show that it is possible to analyse and visualise large sensor datasets efficiently and economically using the power of cloud computing.
We used the Windows Azure platform.
We included Aneka libraries to show it can work in the Windows Azure platform with a view to including it in a substantial way in the future.
One of the significant feature of the application is the use of MEF (Managed Extensibility Framework) library to make the application dynamically extensible.
We also used the Google visualisation library to view sensor location including in 2D and 3D, their values etc.
The sensor data feeds were obtained from Pachube website. We used the data in XML format. The schema was eeml.
The purpose of the application is to allow decision makers to analyse and visualise large sensor datasets quickly and accurately. Only requirement for the user is a computer with a browser and internet access.

In the future:

Ability to analyse and visualize data from any platform. e.g. mobile platforms.
Handle sensor data based on any schema. e.g. SensorML. One way to achieve this is to use XSLT.
Handle any data analytics: Ability to do data analytics on – space junk data, medical data, financial data, weather data etc.
In this project we have just touched on the surface of visualisation aspect of data analytics. Data visualization theory includes amongst many other things, human-computer interaction, data types, chart types, visual variables, visualization techniques, structure of data visualization, navigation in data visualization, colour theory, cognitive theory, and visualization evaluation.
Use the full feature set of Aneka and Azure to develop a powerful and economical data analytics services. Also customers with existing data centres can combine it with the public cloud using Aneka.
Users should be able to save data analytics results in multiple formats.
Provide Single Sign-On (SSO) using WS-Federation protocol.
We can develop a multi-tenant ASP.NET MVC application to support multiple customers. This can allow for customising the application (U.I., functionality provided, costs etc.) for each tenant or type of customers.

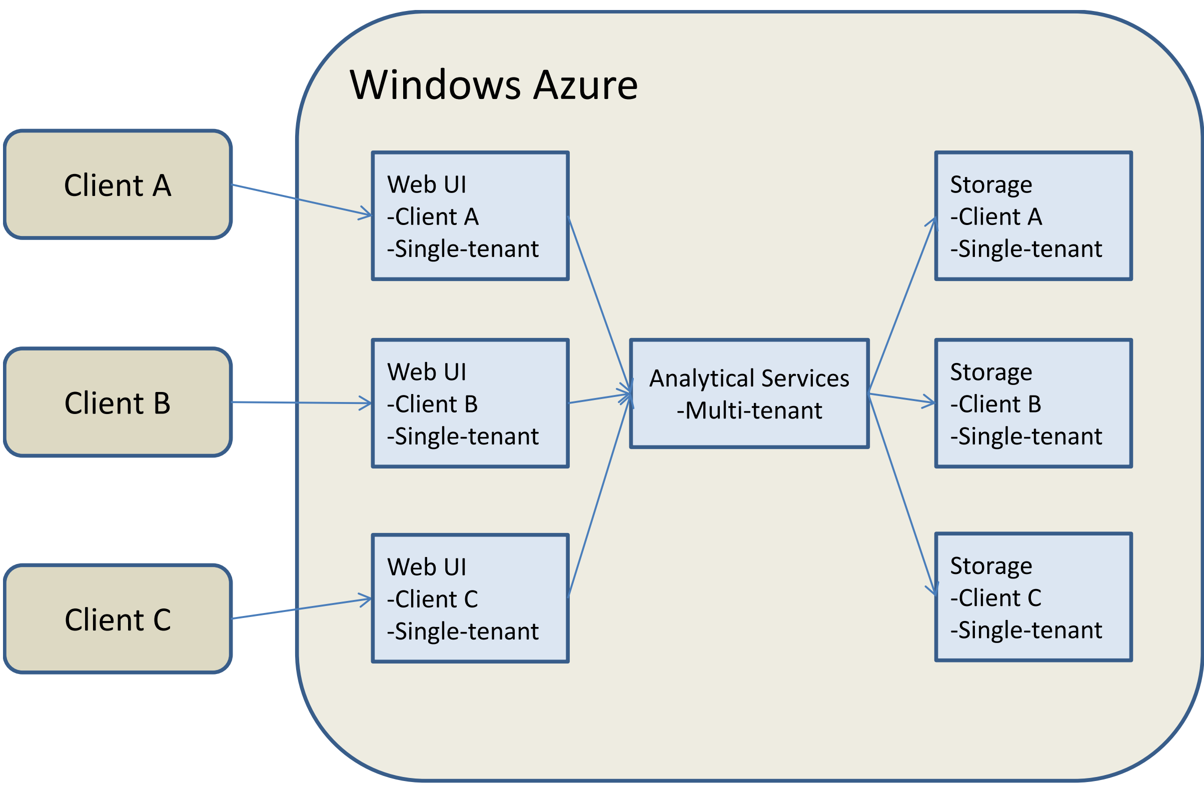


Figure 7.1 Multi-tenant architecture

We can develop a fully-fledged data analytics and visualisation application with Windows Azure and Aneka, for multiple tenants, unlike anything in the market today.

References

[1] A smarter planet: http://asmarterplanet.com/

[2] Internet of Things: http://www.itu.int/osg/spu/publications/internetofthings/

[3] CISCO Internet of Things: http://blogs.cisco.com/news/the-internet-of-things-infographic/

[4] http://en.wikipedia.org/wiki/Sensor

[5] WSN: http://en.wikipedia.org/wiki/Wireless_sensor_network

[6] http://www.azonano.com/article.aspx?ArticleID=1281

[7] Wireless Sensor Networks, Feng ZHAO, Leonidas GUIBAS

[8] Application Architecture Guide, MS Patterns & Practices

[9] Developing Applications for the Cloud, MS Patterns & Practices

[10] MEF : http://msdn.microsoft.com/en-us/library/dd460648.aspx

[11] DavidChappel Associates: The Windows Azure Programming Model

[12] SensorGrid: Integrating Sensor Networks and Grid Computing by Chen-Khong Tham and Rajkumar Buyya

[13] A Survey on Facilities for Experimental Internet of Things Research by Alexander Gluhak, University of Surrey, Srdjan Krco, Ericsson Serbia, Michele Nati, University of Surrey, Dennis Pfisterer, University of Lübeck, Nathalie Mitton and Tahiry Razafindralambo, INRIA

[14] Aneka flyer - http://manjrasoft.com/