



Implementation of a Log Agent in Microsoft Azure

and packaging it to Azure Marketplace.

Implementering av en Log Agent i Microsoft Azure
och paketering till Azure Marketplace

Michael Bui
Magnus Pedersen

Faculty: Faculty of Health, Science and Technology

Subject: Computer Science

Points: 15hp

Supervisor: Kerstin Andersson

Examiner: Donald F. Ross

Date: 2015-05-22

Implementation of a Log Agent in Microsoft Azure

and packaging it to Azure Marketplace.

Magnus Pedersen

Michael Bui

This thesis is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Magnus Pedersen

Michael Bui

Approved, 01/06/2015

Opponent:

Advisor: Kerstin Andersson

Examiner: Donald F. Ross

Abstract

Cloud computing is still in an early stage of development and Microsoft is now investing considerable amount of resources in the cloud. Microsoft Azure is a cloud platform developed by Microsoft and it is continuously evolving, new features are constantly being added and old features are being updated.

Integration Software, which is a company that focuses on products for system integration strongly believes that cloud-based solutions will have a significant impact on their future. This is why selling and developing solutions and services for the cloud are strategically important for them.

The objective of this dissertation is to investigate Microsoft Azure in general and Azure Marketplace in particular. This investigation consisted of an implementation of a Microsoft Azure application and integrating this application with Azure Marketplace and evaluating the expenses for running the application.

The purpose for this project is to gain practical experience and to work with new techniques and help Integration Software better understand Azure Marketplace.

The application is a Log Agent which fetches data from an external source and resends the data to an external party (Integration Manager). Our first intention was to package and deploy the application to a newly updated Azure Marketplace. The new Azure Marketplace was never released during this dissertation so we decided to deploy the application to the existing version of Azure Marketplace. This was however not fully successful. We encountered some problems in successfully deploying the application to Azure Marketplace.

The evaluations for the cost of running an Azure application were not carried out due to lack of time.

Keywords: Microsoft Azure, Azure Marketplace, cloud computing, cloud services, logic apps

Acknowledgements

We would like to thank our supervisor Robert Mayer at Integration Software for all the discussions, supervising and setup of development environment. Also a big thanks to our dissertation supervisor Kerstin Andersson at Karlstad University who read, corrected and gave insight on numerous revisions of this dissertation. We would also like to thank Microsoft for letting us use their infographics. Lastly, we would like to thank Integration Software as a company for supplying a workstation, the necessary equipment and giving us the opportunity to work with them.

Table of contents

Introduction

[1.1 Background and purpose](#)

[1.2 Objective](#)

[1.3 Approach](#)

[1.4 Disposition](#)

Background

[2.1 What is the cloud?](#)

[2.1.1 Why is it called “the cloud”?](#)

[2.1.2 Cloud service layers](#)

[2.1.3 The benefits and disadvantages of cloud computing](#)

[2.2 Overview of techniques and tools used](#)

[2.2.1 Development tools](#)

[2.2.2 Techniques and standards](#)

[2.3 Microsoft Azure](#)

[2.3.1 Microsoft’s Data centers](#)

[2.3.2 Cloud Services](#)

[2.3.3 Azure storage](#)

[2.3.4 Azure Marketplace](#)

[2.3.5 Azure Service Bus Relay](#)

[2.3.6 API Management Portal](#)

[2.3.7 Azure SQL Databases](#)

[2.3.8 Azure Web App](#)

[2.3.9 Integration Manager](#)

[2.4 Overview of the systems](#)

[2.5 Chapter summary](#)

Application design

[3.1 Technical preparations](#)

[3.2 Application design](#)

[3.2.1 Configurations](#)

[3.2.2 The Cloud service](#)

[3.2.3 Azure Storage](#)

[3.2.4 Plugins](#)

[3.2.5 Service Bus Relay](#)

[3.2.6 Researching Azure Marketplace](#)

[3.3 How does the application work?](#)

[3.4 Alternative design](#)

[3.5 Chapter summary](#)

Project implementation

[4.1 The project components and code structure](#)

[4.2 CommonLibrary](#)

[4.2.1 Configuration and ConfigurationHandler](#)

[4.2.2 PluginHandler](#)

[4.2.3 StorageHandler](#)

[4.2.4 Subscription](#)

[4.3 Plugin](#)

[4.3.1 Event class](#)

[4.3.2 Plugin Example - Currency Plugin](#)

[4.4 The Cloud service](#)

[4.4.1 Web Role](#)

[4.4.1.1 Administrator page](#)

[4.4.1.2 Plugin management](#)

[4.4.1.3 User management](#)

[4.4.1.4 Subscription management](#)

[4.4.1.5 Third party login authentication](#)

[4.4.1.6 Web API](#)

[4.4.2 “Master” Worker Role](#)

[4.4.3 “Grunt” Worker Role](#)

[4.4.3.1 Updating plugins](#)

[4.4.3.2 Processing a configuration](#)

[4.5 Scalability](#)

[4.6 Tests](#)

[4.7 Azure Marketplace](#)

[4.7.1 Integrating our application with Azure Marketplace](#)

[4.8 Chapter summary](#)

Results

[5.1 Log Agent Application](#)

[5.1.1 Comparison to what was planned](#)

[5.2 Integrating the application with Azure Marketplace](#)

[5.2.1 “Old” way vs “new” way of deployment to Azure Marketplace](#)

[5.2.2 Conclusion](#)

[5.3 An existing similar system - Logic Apps](#)

[5.4 Chapter summary](#)

[Conclusions and evaluation](#)

[6.1 Project summary](#)

[6.2 Problems](#)

[6.3 Lessons learned](#)

[6.4 Future work](#)

[References](#)

[Appendix A - Installation of development environment](#)

[Appendix B - GUI](#)

[Appendix C - Source code](#)

List of Figures

[Figure 1. Microsoft Azure Annual Capital Expenditures](#)

[Figure 2. Simplified design overview of the project](#)

[Figure 3. MVC Model](#)

[Figure 4. TDD flow chart](#)

[Figure 5. Locations of the data centers that Microsoft provides as of 2015-02-25](#)

[Figure 6. Overview of a Windows Azure Cloud Service](#)

[Figure 7. Azure Table storage](#)

[Figure 8. Azure Queue storage](#)

[Figure 9. Azure Blob storage](#)

[Figure 10. Microsoft Azure Service Bus Relay](#)

[Figure 11. List of events, sorted by date](#)

[Figure 12. Event details.](#)

[Figure 13. Overview of techniques and tools used and how they are connected](#)

[Figure 14. Finalized detailed overview of the project design and its components](#)

[Figure 15. The interaction between the different components in the Azure storage and cloud service. The left side represents the Azure storage and the right hand side represent the cloud service. This figure expands on the cloud service and storage, depicted in Figure 14.](#)

[Figure 16. Azure storage component structure in Azure Storage Explorer 6](#)

[Figure 17. How Azure Marketplace was said to work with the new updates, “new” way.](#)

[Figure 18. How Azure Marketplace works now, “old” way.](#)

[Figure 19. A part of Integration Manager’s user interface](#)

[Figure 20. Initial application design proposed by Integration Software](#)

[Figure 21. Finalized detailed overview of the project design and its components](#)

[Figure 22. Project structure in Visual Studio](#)

[Figure 23. The connections of the modules to each other](#)

[Figure 24. The classes within the CommonLibrary](#)

[Figure 25. GUI of plugin configuration a user can set](#)

[Figure 26. Administrator menu](#)

[Figure 27. GUI of plugin management](#)

[Figure 28. GUI of user management](#)

[Figure 29. Index page for configurations](#)

[Figure 30. Shared view to create and edit a configuration](#)

[Figure 31. The interaction between the different components in the Azure storage and cloud service. The left side represents the Azure storage and the right hand side represent the cloud service.](#)

[Figure 32. Azure subscription invitations in staging](#)

[Figure 33. Workflow for a logic app. First the recurrence is specified then two http connectors which fetches some data from a website and posts it to another URI.](#)

[Figure 34. Detailed overview of the LogAgent application](#)

[Figure 35. Azure Marketplace, the list of all available application \(including our own staged application\)](#)

[Figure 36. Index page](#)

[Figure 37. View to monitor created configurations](#)

[Figure 38. View for creating a configuration, the edit view uses the same view](#)

[Figure 39. Administrator page/plugin management](#)

[Figure 40. Administrator page/user management](#)

[Figure 41. Login page, with the additional services to login with](#)

List of Tables

[Table 1. Configuration table](#)

[Table 2. AspNetUser table](#)

[Table 3. Subscription tier - message limit](#)

Listings

[Listing 1. Serialized and deserialized dictionary](#)

[Listing 2: Plugin.cs](#)

[Listing 3. Currency.cs](#)

[Listing 4. Controller user authorization](#)

[Listing 5. View user authorization](#)

[Listing 6. Html.BeginForm](#)

[Listing 7. ActionResult UploadPlugin in AdminController.cs](#)

[Listing 8. Method UploadPlugin in PluginHandler.cs](#)

[Listing 9. Microsoft Account Authentication](#)

[Listing 10. Method for basic authentication](#)

[Listing 11. Configuration create view form](#)

[Listing 12. Setting up a Quartz.NET job and scheduling it.](#)

[Listing 13. WorkerRole.cs/Run\(\)](#)

[Listing 14. Method that reloads plugins, if needed.](#)

[Listing 15. Method, ProcessConfiguration](#)

[Listing 16. Example of a negative test](#)

[Listing 17. Configuration.cs](#)

[Listing 18. PluginHandler.cs](#)

[Listing 19. StorageHandler.cs](#)

[Listing 20. Plugin.cs](#)

[Listing 21. CreateWorkItemJob.cs](#)

[Listings 22. EditorTemplates](#)

Chapter 1

Introduction

This dissertation is a part of the Bachelor's degree in computer science which was conducted during spring 2015 with Integration Software and Karlstad University.

Integration Software [\[39\]](#) is part of a corporate group that focuses exclusively on system integration. The corporate group consists of two companies; iBiz Solutions (consulting services in system integration) and Integration Software (products in system integration).

This chapter presents a brief background and the purpose of the project, thereafter the objective and approach of the project is defined. Lastly, the disposition of the dissertation is explained.

1.1 Background and purpose

Cloud computing is still in an early stage of development [\[1\]](#) and Microsoft is now investing considerable amounts of resources (see Figure 1) in the cloud, enabling companies to migrate to cloud solutions. Microsoft Azure is a cloud platform developed by Microsoft and it is continuously evolving, new features are constantly being added and old features are being updated. One of these new features, a primary topic of this dissertation, is Azure Marketplace (explained in Chapter [2.3.3 Azure Marketplace](#)) [\[2\]](#).

Integration Software strongly believes that cloud-based solutions will have a significant impact on their future. They believe that a large portion of their sales will be cloud-based. This is why selling and developing solutions and services for the cloud is strategically important for them. Integration Software wishes to explore the possibilities of hosting and packaging applications on a cloud-based platform such as Microsoft Azure. Integration Software is also interested in how to specify a payment plan and the cost of running a cloud-based application.

Therefore, Integration Software has requested a test application to be created in Microsoft Azure and exploring the possibilities within Azure Marketplace.

The purpose of the project is to gain practical experience in cloud computing and help Integration Software understand Azure Marketplace better.

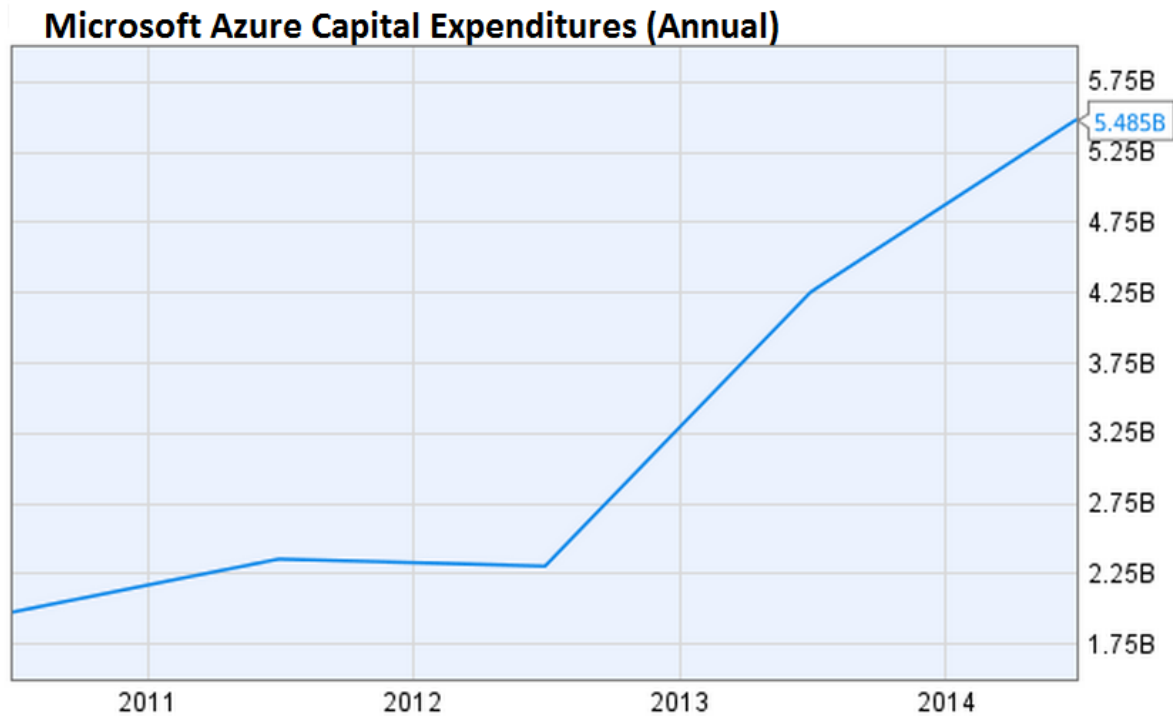


Figure 1. *Microsoft Azure Annual Capital Expenditures* [3]

1.2 Objective

The main objective of this study is to implement a logging agent (see Figure 2 for a basic overview of the application). In short the user will configure the application to get raw data from external sources (e.g. currencies, active directory or service logs from a database). This data is sent to a log tracking system called Integration Manager which is made by Integration Software. Integration Manager will then handle the data (such as filtering) and present it to the user. The logging agent will be implemented with several of the Azure Cloud technologies, many of which are explained in Chapter [2.3 Microsoft Azure](#). The majority of these technologies are well documented by Microsoft and these were studied in detail in order to acquire the required knowledge of the Azure cloud platform.

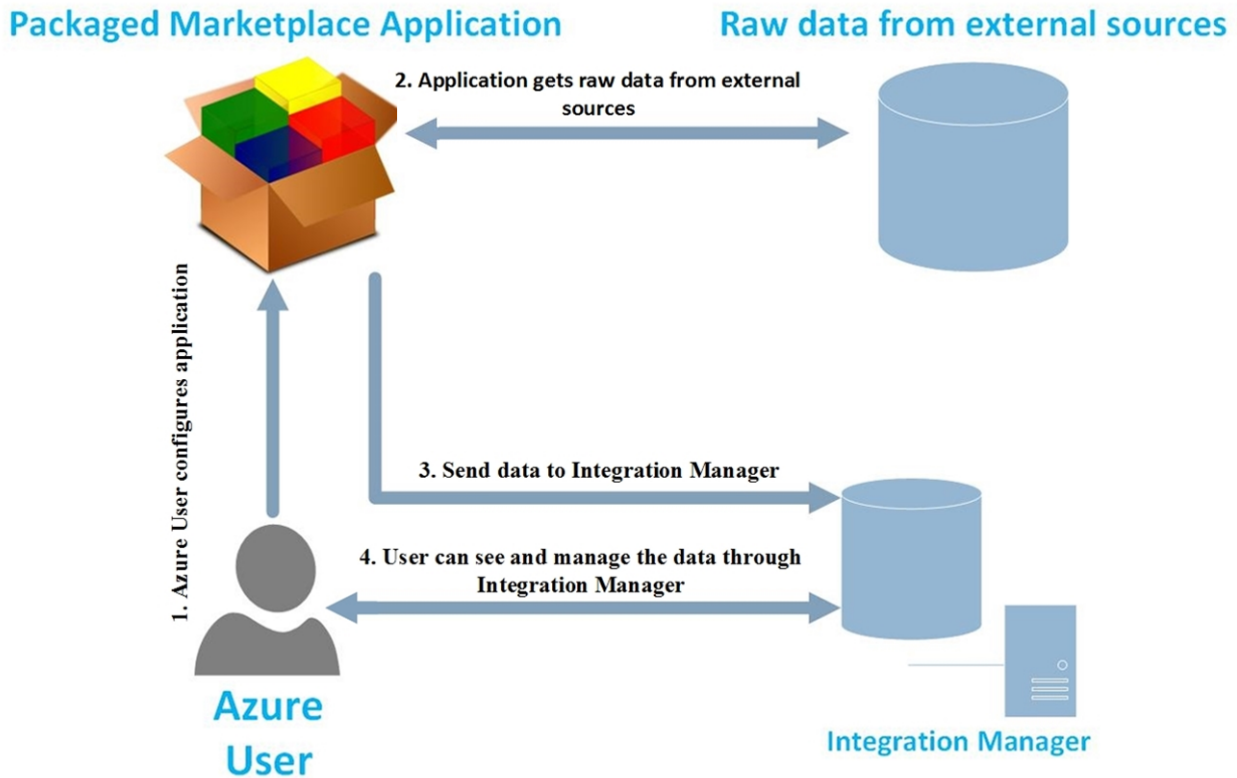


Figure 2. *Simplified design overview of the project*

The second objective is to explore the possibility of packaging the logging agent into an Azure Marketplace application. The intention is to make it possible for a customer to deploy the package on their own Azure account. The reasoning for this is to offload hosting requirements from Integration Manager onto the customer. However, if it is not possible to package the logging agent as an Azure Marketplace application, the application will be hosted by Integration Software on Microsoft Azure and Azure Marketplace will simply act as a portal to the application (explained thoroughly in Chapter [3.2.6 Researching Azure Marketplace](#)). This would make results similar to hosting the application on-premises (running the software on dedicated servers in-house). The benefits and drawbacks of cloud-based hosting are further explained in Chapter [2.1.3 The benefits and disadvantages](#).

1.3 Approach

The first step is to gain a better understanding of the cloud platform and Microsoft techniques. To achieve this studying and testing Microsoft Azure and Microsoft techniques were required. The second step was to evaluate different cloud services and Microsoft techniques and to design the requested test application based on the knowledge gathered from step one.

Step three was to implement the application using the evaluated cloud services and Microsoft techniques.

The last step was attempting to package the application into a Microsoft Azure Marketplace application.

1.4 Disposition

In the beginning of Chapter 2 the cloud is explained. Then an overview of the different development tools and techniques used for this project are introduced and explained to give the reader a general understanding of the tools and techniques used. Then Microsoft Azure and some of its services and applications used in this project are explained. The chapter ends with an illustration on how all the tools and techniques used in this project are connected with each other.

Chapter 3 addresses the project design, where we explain the thought process and preparation for the design of the project. Some crucial parts of the application are also explained such as configurations, the cloud service, storage and plugins. Then we explain how the application works with a concrete example, lastly we discuss some alternative designs for this project.

Chapter 4 addresses the entire implementation of the application. First we introduce the project structure and the different components and modules included in the application. Then we thoroughly explain the implementation of the most crucial parts of the application. After that we discuss some aspects of the application such as scalability and testing. Lastly we explain the integration of the application with Azure Marketplace.

Chapter 5 presents the results of the entire project. The results of the main objectives are explained i.e., the implementation of the application and integrating the application with Azure Marketplace. Finally a similar system is presented, namely Logic Apps. Logic Apps is explained and compared to the application implemented in this project.

Chapter 6 gives a summary of the entire dissertation. The summary reflects on the objectives and purposes given in Chapter 1 and the results of these objectives in Chapter 5. Some topics are evaluated, these are: cloud computing, costs for Microsoft Azure, application development and Azure Marketplace. Then the problems which occurred during this project are explained and lastly we give some personal insights on the lessons learned, hindsight and possible future work for this project.

Chapter 2

Background

This chapter describes concepts, techniques and terminology necessary for understanding the following chapters. First the cloud is explained, then the various development tools and techniques used in this project are explained. Finally Microsoft Azure and some of its applications and services which were evaluated in this project are explained.

2.1 What is the cloud?

The cloud or cloud computing, as it is sometimes referred to, is simply storing and accessing data and programs over the Internet instead of your own physical hard drive. Some popular and extensive cloud services include Google Drive, Microsoft SkyDrive, Dropbox and Apple iCloud. These services are called Software as a Service (SaaS). There are two other main foundations of cloud layers - Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). These layers are explained in Chapter [2.1.2. Cloud service layers](#).

2.1.1 Why is it called “the cloud”?

Simply put, the cloud is a metaphor for communication over Internet. Because of the complexity of Internet, engineers needed an abstract way to represent the Internet without really having to explain all the details about it. Thus “the cloud” became a frequently used term and symbol.

2.1.2 Cloud service layers

Cloud computing is often divided into three main layers. Software-as-a-Service, Infrastructure-as-a-Service and Platform-as-a-Service.

- **Software-as-a-Service (SaaS)**

SaaS is software provided over the Internet. The application runs and executes in the cloud without installation on the actual machine. As mentioned earlier, some typical SaaS applications are Google Drive, Microsoft SkyDrive, Dropbox and Apple iCloud.

- **Infrastructure-as-a-Service (IaaS)**

IaaS provides computing resources such as servers, virtual machines, networking and storage. In Microsoft Azure these computing resources are managed by Microsoft Data Centers (explained in chapter [2.3.1 Microsoft’s Data centers](#)). This means that the

customers avoid having computing resources on their own premises. Examples of IaaS are Microsoft Azure, Amazon Web Service and Google Compute Engine.

- **Platform-as-a-Service (PaaS)**

PaaS is basically an IaaS but the cloud provider also delivers “middleware” such as operating systems, development tools, web server, etc. Examples of PaaS are Windows Azure, Heroku and Google App Engine.

2.1.3 The benefits and disadvantages of cloud computing

The benefits of having a cloud solution are that companies do not need to worry about managing and maintaining facilities and hardware such as servers and virtual machines.

All the data centers in the cloud are maintained and managed by the cloud service provider. Cloud computing has the benefit of being easy to scale. A customer can easily scale, say a virtual machine or a website, either manually or automatically by letting the system scale the service based on the workload of the service. Often referred to as on-demand-performance. Many cloud service providers, such as Microsoft Azure, make setting up virtual environments trivially easy, e.g. installing a complete operating system can be achieved by the click of a button.

By using the cloud, customers only pay for what they use.

E.g, if a company experiences surges of traffic during weekends they may scale up their service during weekends and only pay for the extra performance used during that period.

Data in the cloud is safely stored [\[4\]\[5\]](#) (further explained in [Reliability, redundancy and privacy](#)) on data centers and backing up and restoring the data is much easier than on a physical drive. Data in the cloud is easily accessed, all you need is an Internet connection and you can access the data from anywhere.

The biggest concern and downside with cloud computing is the security aspect. Many people and companies believe that the data stored on the cloud is not secure, although Frank Bauerle, a SaaS Platform Delivery Architect, states: “Your cloud infrastructure is only as secure as you make it” [\[6\]](#). Most cloud platforms offer security measures such as firewalls, network gateways and antivirus. As with other services on the Internet, cloud computing is prone to external hack-attacks and threats. Switching to a cloud-based solution also means that companies may deliver sensitive information to a third-party cloud service provider, which is probably one of the reasons companies do not wish to migrate to cloud solutions - it is a trust issue!

2.2 Overview of techniques and tools used

This subchapter will explain the different development tools, techniques and standards used to develop the Azure Marketplace Application.

2.2.1 Development tools

All of the tools needed to work with Azure applications are supplied by Microsoft.

All the development tools, the IDE (Integrated Development Environment) and Azure accounts required to develop the application were all provided by Integration Software. Alternative IDEs were not considered as there seems to be very few other IDEs with integrated Azure functionality, which simplifies the process of deploying and testing Azure applications. Also worth mentioning is that most of the documentation for developing Azure services is often written with the assumption that Visual Studio is being used as the IDE.

Visual Studio 2013 (VS13) [\[7\]](#) is the development environment used to develop the designated application. Visual Studio is made by Microsoft and it has a broad range of capabilities such as debugging, code completion, integrated version control, syntax checking and testing frameworks for most of the common Microsoft techniques. With the Azure SDK (Software Development Kit) installed it is also possible to directly deploy to Microsoft Azure services and test Cloud applications locally.

Remote Desktop Protocol (RDP) [\[8\]](#) was used to connect to a remote desktop over a network connection. This made it possible to work on a remote desktop on a local computer. Windows Server 2012 R2 [\[9\]](#) is the operating system used on the remote desktop to develop the Azure Marketplace application.

Visual Studio Online [\[10\]](#) is a software development platform which makes it possible to host software projects in the cloud. There is support for several versioning systems, such as git and Team Foundation Server. It is possible to manage workflow by creating backlogs, creating tasks, tracking bugs, tracking progress with agile task boards and automating the build process, in the cloud.

.NET Framework [\[11\]](#) is a software framework by Microsoft. It has a large class library called the FCL (Framework Class Library) and supports language interoperability for certain languages such as C#, C, C++ and Visual Basic. Programs using the .NET framework are executed in a virtual machine, called CLR (Common Language Runtime). The FCL provides a large range of useful features, such as user interface, database connectivity and web application development.

2.2.2 Techniques and standards

In this subchapter various techniques and standards used for this project are explained in detail.

Windows Communication Foundation (WCF) [\[12\]](#) is part of the .NET framework and consists of a runtime and a set of Application Program Interfaces (APIs) for making connected service applications. WCF services can be used to create systems that send messages between a service and a client. WCF makes it possible to remotely call predefined functions which are defined in a service contract, which specifies the signature of the service and the data it exchanges. The service contract, which is implemented by the service, determines how the responses of the function calls are created. In this project our designated application sends data via a Service Bus Relay (Explained in Chapter [2.3.7 Azure Service Bus Relay](#)) to a WCF service that Integration Manager exposes.

Representational State Transfer (REST) [\[40\]](#) is a software architecture pattern which describes how web services manages machine to machine-communication. Systems communicating with REST use the same HTTP verbs (such as GET, POST, PUT and DELETE) that are used by web browsers to retrieve and send web pages and data. The Web API implemented in this project uses REST.

Model-View-Controller (MVC) [\[13\]](#) is a design pattern used in system development. It separates an application into three interconnected parts (Model-Data, View-Presentation, Controller-User interaction) (see Figure 3). The user web interface implemented for the application uses this design pattern (explained in chapter [3.3.2 The Cloud service](#)).

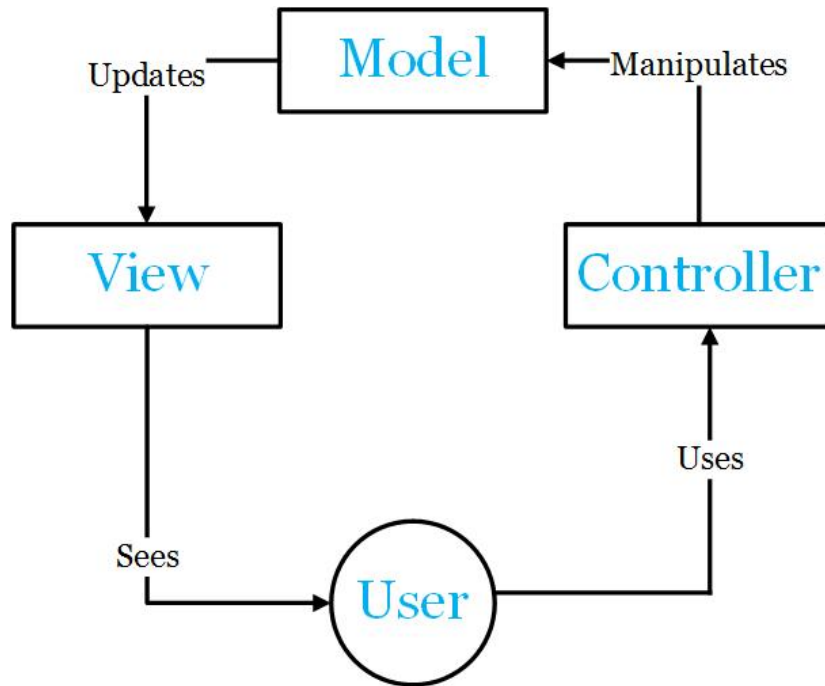


Figure 3. *MVC Model*

Test-driven development (TDD) [14] is a software development method which focuses heavily on automated unit tests. As shown in Figure 4, this technique advocates the developer to write failing unit tests before writing actual programming code. The developer then writes enough code to pass the test and finally refactors the code. This is done in short iterations. Before the code can be checked in the developer must ensure that all unit tests are successful. TDD was used in this project to ensure high quality, stability and test coverage.

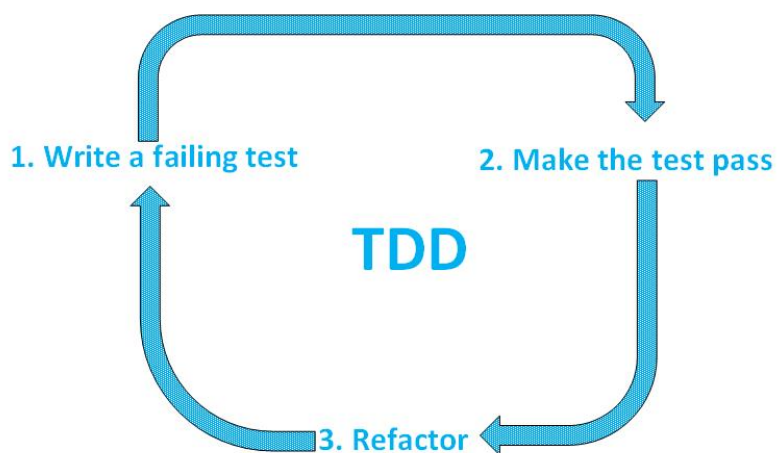


Figure 4. *TDD flow chart*

2.3 Microsoft Azure

This subchapter will describe Microsoft Azure and some of the applications and services offered by the Microsoft Azure cloud platform which were evaluated in this dissertation.

Microsoft Azure [15][16] is a cloud-based platform and infrastructure. Microsoft Azure can be used to build, deploy and manage applications and services. The platform is hosted and managed on one of many global Microsoft Datacenters (explained in the next subchapter). Microsoft Azure provides a variety of PaaS and IaaS and Microsoft states that "Azure supports almost any operating system, programming language, tool and framework" [16].

As previously mentioned in Chapter 2.1.3 [The benefits and disadvantages of cloud computing](#), using Azure is economical and scalable [16]. The reason for this is that Azure can quickly scale up or scale down services and applications to match the workload. You only pay for what you use, sometimes called per-minute billing and on-demand-performance.

Visit the link¹ in the footnote for a complete overview of the Microsoft Azure cloud platform.

2.3.1 Microsoft's Data centers

The entire cloud platform is hosted on so called "data centers" which are located all over the world. As of March 2015 Microsoft Azure is available in 140 countries, supports 10 languages and 19 currencies [17] - This is a result of Microsoft's \$15 billion investment [17] in global data center infrastructure. The Azure cloud platform is a rapidly growing network of data centers and Microsoft is continuously investing in the latest infrastructure technologies [17]. The main focus is to maintain high reliability, operational excellence, cost-effectiveness and environmental sustainability. Figure 5 shows the locations of all the global data centers.

¹ <http://azure.microsoft.com/en-us/documentation/infographics/azure>



Figure 5. *Locations of the data centers that Microsoft provides as of 2015-02-25* [\[17\]](#)

Reliability, redundancy and privacy in the cloud

The Azure cloud platform is always up and running, it offers 99.95% availability and uses the same enterprise-tested platform that powers Skype, Office 365, Bing and Xbox [\[16\]](#). Azure also offers a service-level agreement (SLA), 24/7 tech support and round-the-clock-service [\[17\]](#).

With data redundancy Azure ensures high reliability, durability and availability. Azure offers three storage replication options [\[26\]](#) :

- Locally-redundant storage (LRS): Three copies of your data are stored within a single data center in a single region. This provides protection against normal hardware failures, but not from failure of that single data center.
- Zone-redundant storage (ZRS): Three copies of your data are stored across two to three different data centers within one or two regions. This provides higher durability than LRS since it ensures that the data is protected even if one data center happens to fail.
- Geo-redundant storage (GRS): This is the default option for data storage. The data is replicated in a total of six times. The data is replicated three times within the primary region and three times in a secondary region, hundreds of miles away from the primary region. In the case of failure on the primary region Azure storage will failover to the secondary region. Since it is very unlikely that all the data centers in those regions crash at the same time it means that the data is always up, and at worst at a small delay.

Microsoft Azure is in compliance with several compliant programs including the E.U. Data Protection Directive (95/46/EC) and ISO 27001/27002[27]. Microsoft Azure Trust Center was created to manage security and privacy issues regarding the cloud platform (see [28] [Microsoft Azure trust center](#) for more information).

2.3.2 Cloud Services

Microsoft Azure Cloud Service [18] is a PaaS which can be used to create highly available, scalable and cheap to operate cloud applications and APIs in a matter of minutes.

As shown in Figure 6, a Cloud Services application consists of two kinds of roles, web role and worker role. Roles can be viewed as virtual machines which run a variant of Windows Server. Web roles run a variant of Windows Server with IIS (Internet Information Service) while worker roles run the same operating system without IIS. An IIS is a web server created by Microsoft. These virtual machines can be configured. However a cloud service does not work the same way as a virtual machine. With virtual machines you have to provide and set up the environment which your application will run in, while for cloud services the environment is provided for you.

The environment is managed automatically by Azure, unlike a virtual machine which requires maintenance, e.g. patching the operating system. A cloud service can scale by increasing the number of instances for any particular role. Requests are balanced across instances by a load-balancer. A cloud service should be written in a way so that if one instance of a role fails it still keeps running. This is achieved by not saving state to file storage and instead using shared external storage such as blobs and tables (explained in Chapter [2.3.3 Azure storage](#)).

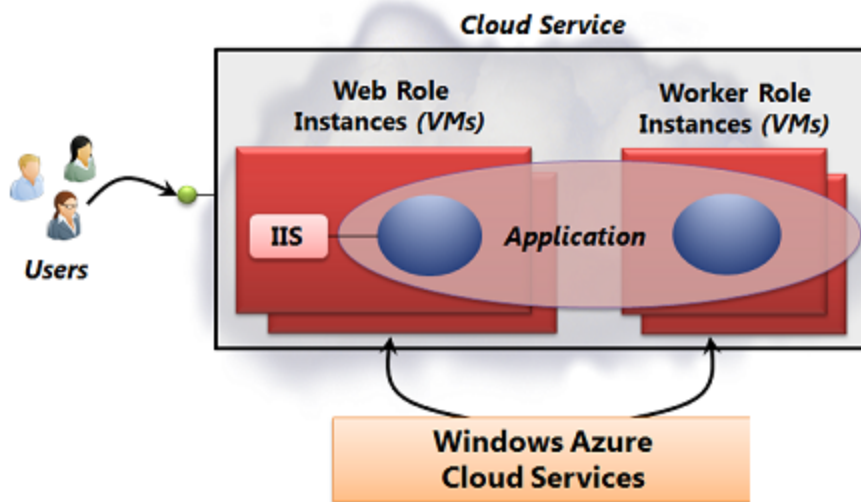


Figure 6. Overview of a Windows Azure Cloud Service [\[18\]](#)

2.3.3 Azure storage

Azure storage [\[19\]](#) provides highly available storage for applications running both in the cloud and on-premises. As with most Azure services, Azure storage is highly scalable and automatically load balances based on traffic, this means that depending on how frequently data is being accessed the load is distributed across several servers in different locations. There are four different types of storage. Table storage, Queue storage, Blob storage and File storage.

Table storage uses NoSQL key-attribute to store data. NoSQL is an alternative method of storing and retrieving data to relational databases. Creating an Azure Table storage does not require hosting a Database Management system and as it scales depending on usage, the user only pays for how much table storage is used. Figure 7 below shows how data is structured in an Azure Table storage. Table storage was used in this project to save user configurations and information, such as login and access controls.

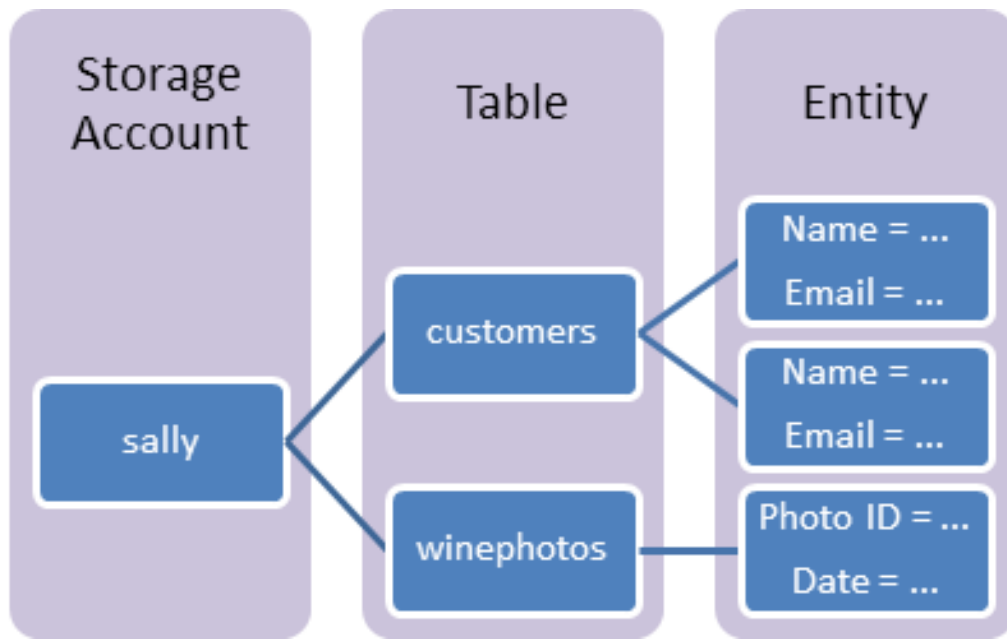


Figure 7. *Azure Table storage* [20]

Queue storage is a queue consisting of any type of messages which are accessed via HTTP or HTTPS. Cloud Services can have multiple roles which run in parallel, so to coordinate and exchange messages between these multiple roles a Queue storage can be used. Accessing a Queue storage is an atomic operation. Figure 8 shows how a Queue storage can be used to execute operations on images. Queue storage was used in this project to keep track of the version of the plugins and communicating between the different roles.

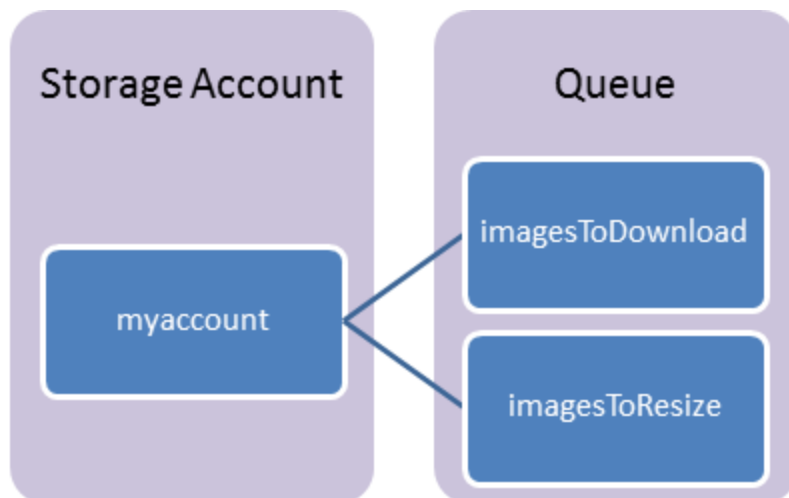


Figure 8. *Azure Queue storage*

Blob storage is used to store and access any kind of unstructured data. Blobs are organized by creating containers, which can be used to set security policies, e.g. to restrict access to a certain container. There are 2 kinds of blobs, block blobs and page blobs. Block blobs are used to store documents or media files. A page blob is a virtual hard drive, which supports random read and write access. Blob storage also provides atomic operations, for example when updating a blob the old version of that blob continues to be served until the update is complete, to avoid serving a corrupted file. Blob storage was used in this project to save the plugin dynamic-link library (DLL) files. Figure 9 illustrates Blob storage structure.

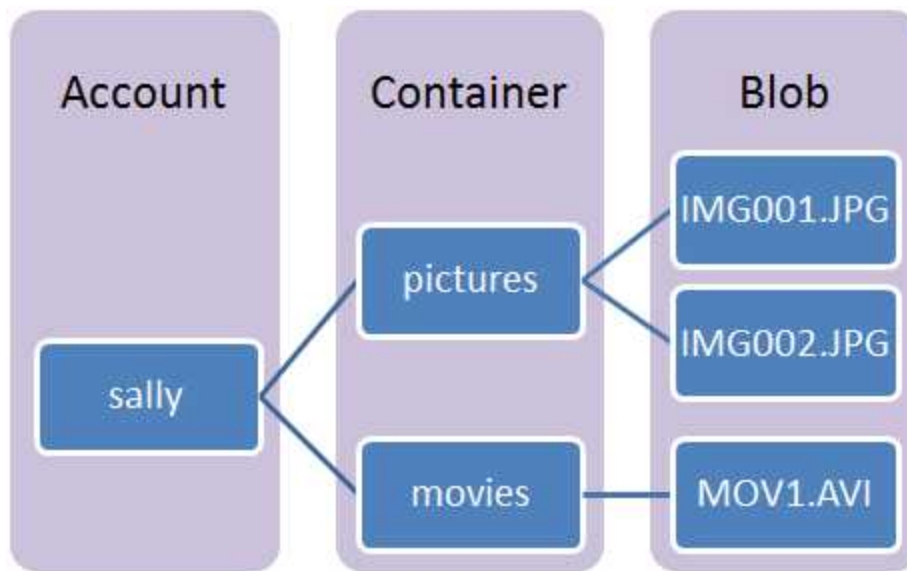


Figure 9. *Azure Blob storage*

File storage is used by applications as a shared storage using the SMB 2.1 protocol [\[41\]](#). File storage is often used to lift existing on-premises applications, which rely on shared file storage to the cloud.

File storage was not used in this project as blobs provided all the functionality required.

2.3.4 Azure Marketplace

Microsoft Azure Marketplace [\[21\]](#) (formerly Microsoft Azure Datamarket) is an online marketplace where independent software vendors (ISVs) can publish and distribute their services and applications to Azure customers all over the world. It is loosely comparable to Google Play or Apple's App Store for mobile applications. More about Azure Marketplace will be explained in Chapter [3.2.6 Researching Azure Marketplace](#) and [4.8 Azure Marketplace](#).

2.3.5 Azure Service Bus Relay

A Microsoft Azure Service Bus Relay [\[22\]](#) makes it possible to expose a WCF service located on-premises, without the need to open a firewall. So instead of connecting to an on-premises server it is possible to connect to a listening service bus running in Azure. Figure 10 illustrates this. This makes it possible to expose an on-premises WCF service to other Azure applications while maintaining high security due to the security mechanisms Azure service bus uses. The Azure Service Bus Relay was used in this project to send data to an external source (Integration Manager).

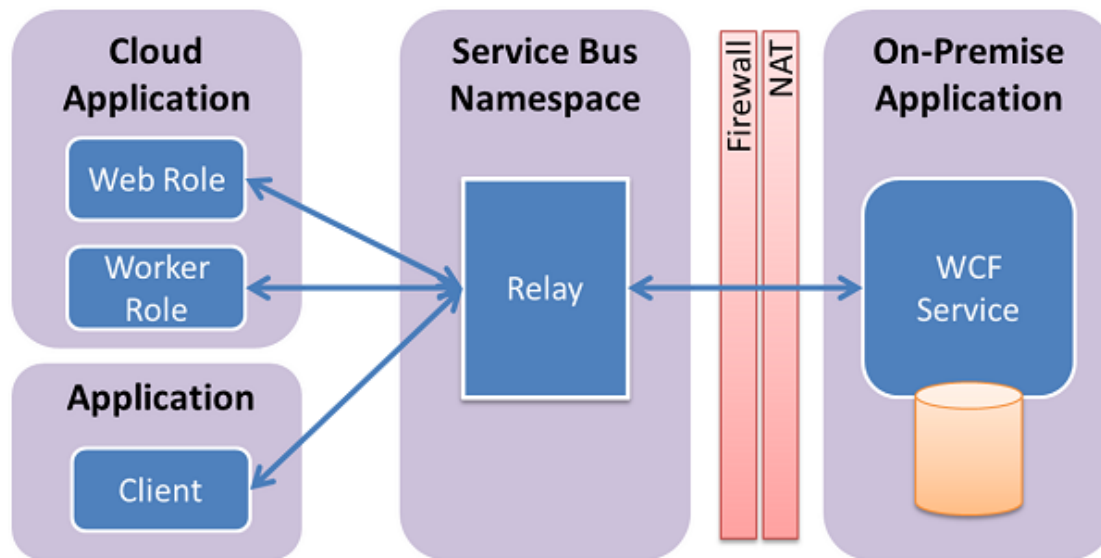


Figure 10. *Microsoft Azure Service Bus Relay*

2.3.6 API Management Portal

API (Application Programming Interface) Management Portal [\[23\]](#) is a tool used to publish and manage APIs, which can be used to, e.g. limit access to an API by limiting the number of queries or only serve registered users. This tool also makes it possible to identify how much an API is used, which can be useful for charging users. This Azure service was evaluated and tested but not used in this project.

2.3.7 Azure SQL Databases

Microsoft Azure SQL Server [\[24\]](#) is a relational database management system hosted on the Azure cloud service. It is based on Microsoft SQL Server which uses a variant of SQL called Transact SQL (or T-SQL) for making queries. This Azure service was evaluated and tested but not used in the development of this project.

2.3.8 Azure Web App

With Azure Web App [\[25\]](#) it is possible to setup and deploy websites. There are several frameworks/languages which Azure Web App supports, such as PHP, Python, C# and Node.js. It offers load balancing and “AutoScale”, which automatically scales your website based on load or according to a schedule, e.g. increase the website’s service capacity to withstand the predicted peak hour at 20:00 Friday. Azure Web App have so called Web jobs, which makes it possible to run scripts in the background, similar to how running shell-scripts would work if the website was hosted on a dedicated server. These scripts either run continuously or on a schedule. Web jobs also make it easier to work with certain Azure methods, such as Azure blobs which are used to store data. This Azure service was evaluated and tested but not used in the making of this project.

2.3.9 Integration Manager

This is only a brief introduction to Integration Manager. Integration Manager has several other qualities, in this subchapter only the features used in this project are explained. See this reference for more information about Integration Manager. [\[43\]](#)

From Integration Manager’s webpage:

“Integration Manager offers a way of working with system integrations like never before by addressing your need to save time and money. When complex systems communicate with each other, Integration Manager logs and gives you full transparency and control of your information. Independent of platform, the tool provides full data logging, smart monitoring, a flexible reporting system and a graphical repository model to show you how your system integration connects and behaves.” [\[39\]](#)

Integration Manager exposes a WCF service which can be used to send logs. This data can then be filtered and monitored using Integration Manager’s web-GUI. Figures 11 and 12 show a listing of events and the details of an event.

Integration Manager 3.1.0.25

Integration Software | Demo

Welcome, BIZmapa

Dashboard

Log

Monitor

Repository

Administration

With selected

Copy API URL

Print

Export to CSV

Save result as: zip

102550100200

«1234567...32»

Log Date Time	State	Message Type	End Point	Direction	Log Text	Action
2015-04-21 13:12:00		String logger	KauTest		Log text	Action
2015-04-21 13:11:59		String logger	KauTest		Log text	Action
2015-04-21 13:11:30		String logger	KauTest		Log text	Action
2015-04-21 13:11:30		String logger	KauTest		Log text	Action
2015-04-21 13:11:00		String logger	KauTest		Log text	Action
2015-04-21 13:11:00		String logger	KauTest		Log text	Action
2015-04-21 13:10:30		String logger	KauTest		Log text	Action
2015-04-21 13:10:30		String logger	KauTest		Log text	Action
2015-04-21 13:10:00		String logger	KauTest		Log text	Action
2015-04-21 13:10:00		String logger	KauTest		Log text	Action
2015-04-21 13:09:30		String logger	KauTest		Log text	Action
2015-04-21 13:09:30		String logger	KauTest		Log text	Action

Figure 11. *List of events, sorted by date*



Additional Fields Values		Context Values	Repository Model
Key	Value		
#	25783		
Log date time	2015-04-21 13:12:00		
Integration			
Service			
System			
End Point	KauTest		
End Point URI	http://kautest.servicebus.windows.net/currencylogger/1.0		
Message Type	String logger		
Original Message Type	String logger		
Direction	 Send		
Size	58 bytes		
Log text	Log text		
Processing machine	Azure		
Process	IM Log Agent		
Processing state	 Processed		
Processing module	StringLogger		

Figure 12. *Event details*

2.4 Overview of the systems

Figure 13 illustrates the overview of the techniques and tools used in this project and how they are connected with each other. Figure 13 is grouped in three systems, development system, Microsoft Azure and external system. The development system represents us developers working on-premises (locally) to test what we later deploy to the cloud platform Microsoft Azure. Some external systems are shown on the right hand side.

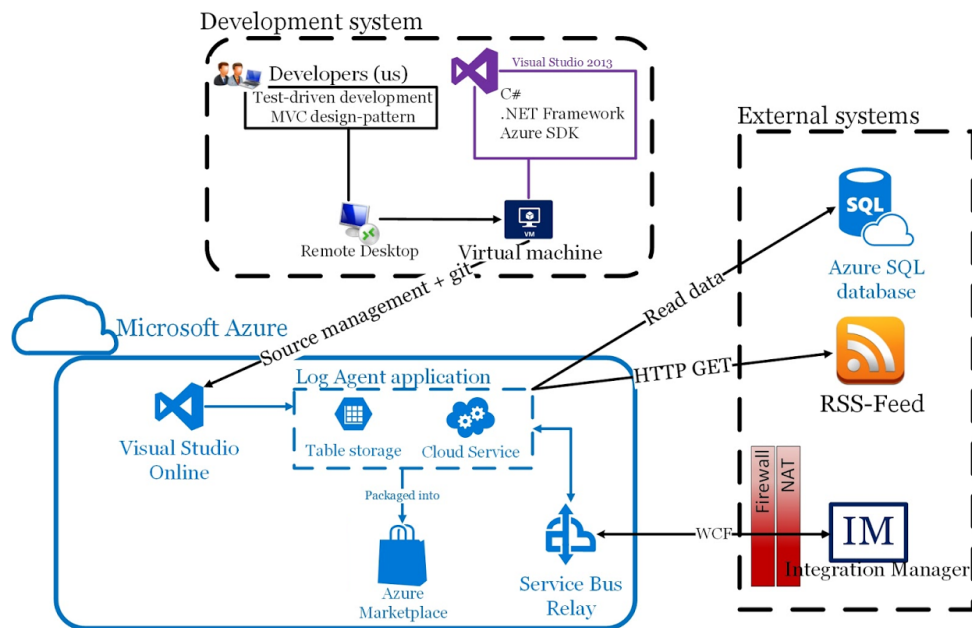


Figure 13. *Overview of techniques and tools used and how they are connected*

2.5 Chapter summary

In this chapter various concepts and technologies were described. The following topics were mentioned:

- The cloud and the three layers of cloud computing, SaaS, IaaS, PaaS. Their benefits and downsides were also discussed.
- Microsoft Azure and several of its components are explained, such as Cloud Services, Azure storage, Azure Service Bus Relay and Azure Marketplace.
- The various tools and techniques used in this project were explained and lastly an overview of all the tools and techniques was shown by a figure.

In conclusion this resulted in an overview breaking it into three major systems;

- Development system which includes Visual Studio along with .NET, Azure SDK and C#. And us developers with the TDD and MVC mindset.
- Microsoft Azure which includes Visual Studio Online where the application is built and deployed to run the various services (Cloud Services, Azure Table storage and Azure Service Bus Relay). This is then supposed to be packaged into Azure Marketplace.
- External systems which includes Integration Manager which our application sends the data to with the help of Azure Service Bus Relay and WCF. Other external sources also includes the sources where the application gets the raw data from. These sources can be anything from RSS-feeds to service logs from a database.

Chapter 3

Application design

This chapter explains the thought process and preparations for evaluating the different Microsoft Azure services and applications used to design the log agent. Then the design of the application and its components are thoroughly explained.

3.1 Technical preparations

Before the development of the application began, studying and testing different Azure and Microsoft techniques were required in order to gain a better understanding. First an experiment involving a Windows service exposing a WCF-service was performed. This gave a basic understanding on how Windows services work. This is relevant because a Windows service works similarly to how a worker role works within a cloud service. In order to gain a better understanding of SQL databases some experimenting and reading was necessary.

After reading up on the local counterpart of cloud-techniques, we moved on to the corresponding techniques for the cloud platform Microsoft Azure. The techniques, which were previously mentioned in [2.3 Microsoft Azure](#), were investigated, tested and evaluated. These Azure services and applications are well documented, e.g. by pictorial step-by-step tutorials showing how to implement and use these techniques. The relevant Azure techniques used in this project were:

- Cloud Services [\[29\]](#)
- Service Bus Relay [\[30\]](#)
- Azure storage [\[31\]](#)
- Microsoft Azure Marketplace [\[21\]](#)

Other Azure services and applications which were investigated, tested and evaluated but not used in the making of the application include:

- Web App [\[25\]](#)
- SQL Database [\[32\]](#)
- API Management portal [\[33\]](#)

See the references of each Azure technique for documentation on planning, developing, deploying and managing the corresponding Azure service/application. See [Appendix A: Installation of development environment](#) for installation of the development environment.

3.2 Application design

After evaluating and discussing the project design we came up with a solution to implement the plugins/API with the help of independent plugins which are stored in an Azure Blob storage. The cloud service consists of two worker roles, the reason for this is so that the application can scale more easily (explained in depth in Chapter [4.5 Scalability](#)). As mentioned in Chapter [2.3.2 Cloud Services](#) cloud services scale by increasing instances of a role. Instances and roles run independently so to communicate they use cloud queues and shared storage, in this case table storage.

Figure 14 illustrates the finalized project design, its components and how they are connected.

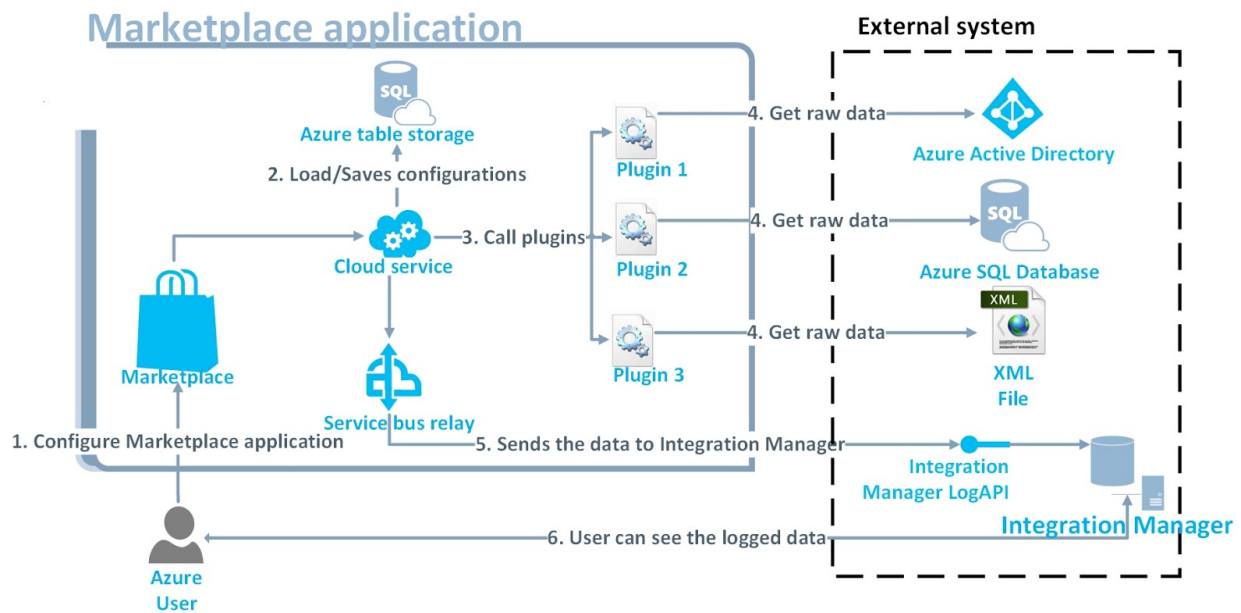


Figure 14: Finalized detailed overview of the project design and its components

The major components of the application are:

- The Cloud service which consists of three components; a web role (acts as the Graphical User Interface), a master worker role (which schedules work and puts it in a job queue) and a worker role (a grunt worker that executes the jobs in the job queue)
- Azure storage, containing data such as configurations, user information, plugins, job queue, version queue.
- Plugins.
- Integration Manager, where all the data is presented to the user. Explained in Chapter [2.3.9 Integration Manager](#).
- Azure Service Bus Relay
- Azure Marketplace

Figure 15 shows detailed information of the interaction between the Azure storage and cloud service and its components. All these components will be thoroughly explained, how they work and how they interact with each other.

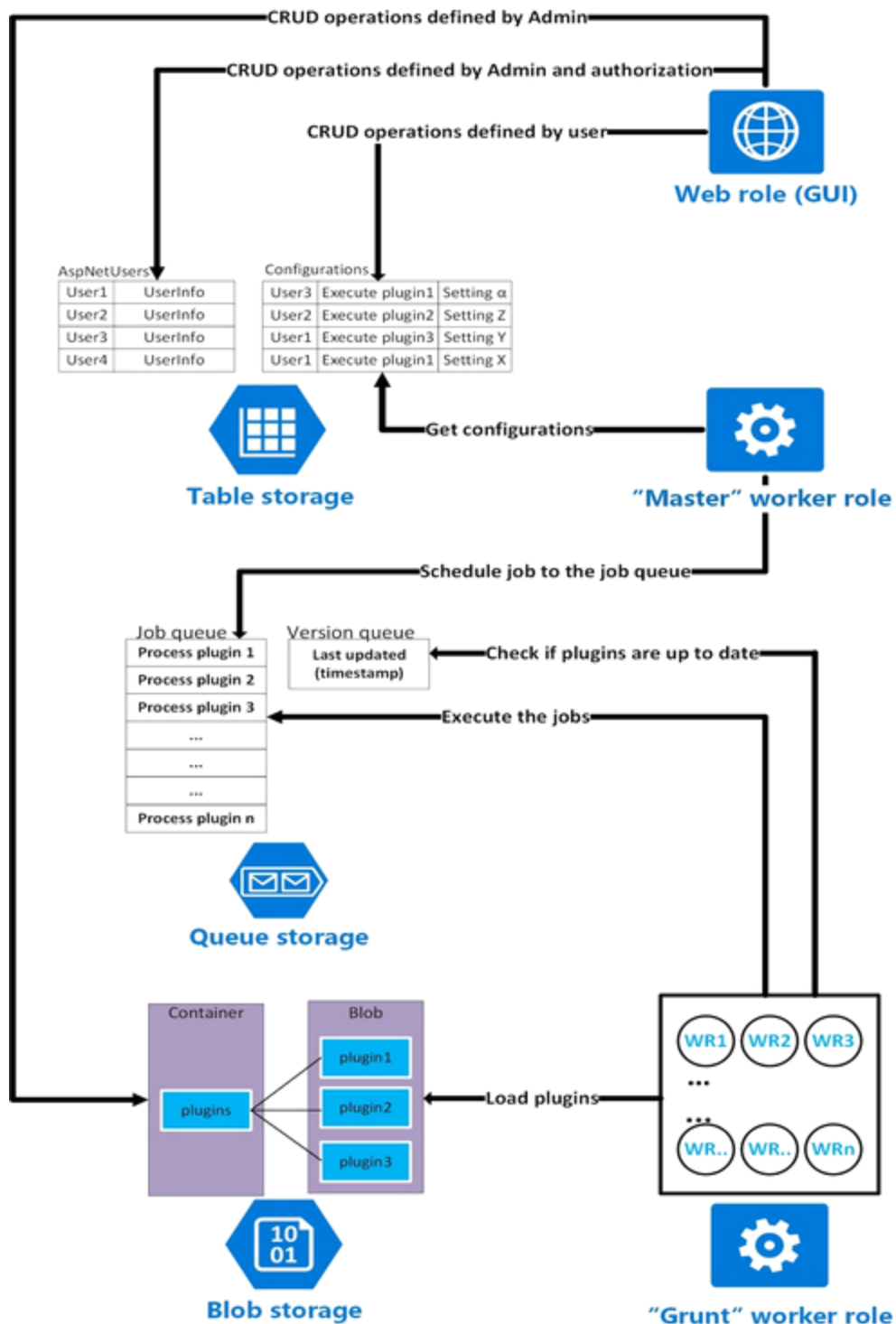


Figure 15. The interaction between the different components in the Azure storage and cloud service. The left hand side represents the Azure storage and the right hand side represents the cloud service. This figure expands on the cloud service and storage, depicted in Figure 14.

3.2.1 Configurations

A Configuration represents in what way logging will be done. A configuration is stored in an Azure Table storage. Configurations can be created by the user via the GUI.

A configuration consists of:

userID - The user who created the configuration, stored as the PartitionKey in the table.

id - A unique ID created with a Globally unique identifier (GUID) method, stored as RowKey in the table.

createdDate - A DateTime object that saves the time when the configuration was created.

cronSchedule - A Cron expression in the Quartz.NET library. This is used to schedule the configuration. For documentation about how a cron expression works see this reference [\[38\]](#).

plugin - The specified plugin to be executed.

pluginConfiguration - A key-value array (dictionary). Optional field, this is used to specify configuration specific information to the plugin. E.g. specifying which currencies the currency tracking plugin should log.

reschedule - This is a flag that tells if the plugin has been updated. If set, the plugin will be reloaded.

eventKeyValuesSerialized - This is used to apply specific event key/values to the events which are sent to Integration Manager.

3.2.2 The Cloud service

As shown in Figure 15, the cloud service consists of three components; web role, “master” worker role and “grunt” worker role.

Web role - Graphical user interface

The web role is the frontend or GUI (graphical user interface) in the form of a website. It exists to provide an interface for the user to set up the cloud service such as CRUD operations (create, read, update, delete) for configurations, plugins, and users. See [Appendix B - GUI](#) for images of the GUI. The user may also monitor the configurations that are currently running through the GUI.

There are some restrictions as to what a user may have access to. The web role restricts users so that only the administrator may have access to the “administrator page”, where CRUD operations to user accounts and plugins can be made. If you are not an administrator you can only manage your own configurations. An administrator has access to every user’s configurations.

The “master” worker role

The “master” worker role delegates work to the “grunt” worker roles. It does this by serializing configurations and when their schedule is due; adding them to a cloud queue called the job queue. The “grunt” worker roles polls the job queue for configurations on short intervals and processes the configuration and updates the corresponding user’s usage statistics based on the resources used to process the configuration.

The “grunt” worker role

As mentioned earlier the “grunt” worker role does all the processing and fetching of data. The “grunt” worker role pops configurations from the job queue and looks at which plugin the configuration specifies and then executes that plugin. Plugins can be loaded at any time during runtime, so to keep all the worker roles’ plugins synchronized a version queue is used. The version queue is posted to whenever a plugin has been added or updated through the GUI.

3.2.3 Azure Storage

The Azure storage consists of three components; table storage, queue storage and blob storage. Figure 16 shows the structure of these components and what they contain.

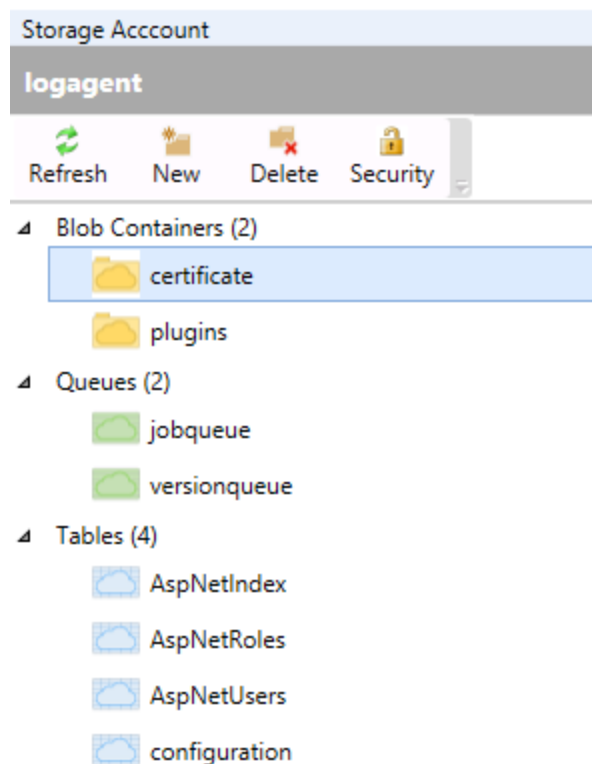


Figure 16. *Azure storage component structure in Azure Storage Explorer 6* [\[51\]](#)

Table storage

As shown in Figure 16, the table storage consists of four tables. But only two of the tables are of interest, the configuration table and `AspNetUsers`. The other two tables are created automatically by a library called `ElCamino`, these tables may be used in the future if the system is to use role based access control. The configuration table consists of information about how a scheduled configuration runs (see [4.2.1 Configuration](#)). Table 1 is an example of how data is stored in the configuration table.

Table 1. *Configuration table*

PartitionKey	RowKey	createdDate	cronSchedule	plugin	pluginConfigSerialized	reschedule
bui.michael@	1c5fe1e2-4f	2015-03-23	* /45 * * ? * MON,TU	StringLogger	textString=123	False
magnus192@	9739af4c-7f	2015-03-23	0 * /30 * ? * MON,TU	StringLogger	textString=mvngpls	False

The `AspNetUsers` table consists of the registered users that have access to the application. Only an administrator may register a new user. The `AspNetUser` table also has two columns (`messageCount` and `subscription`) which are used to restrict the user from sending too many events based on their subscription tier (explained in Chapter [4.2.4 Subscription](#)). Table 2 shows how data is stored in the `AspNetUsers` table.

Table 2. *AspNetUser table*

UserName	Access	Email	EmailC	Lockou	PasswordHash	PhoneNum	SecurityStamp	TwoFactorEnabled
bui.michael@hotmail.com	0	bui.michael@hotmail.com	False	True	AAjraelOYQwti	False	147e1ce5-a516-	False
magnus192@hotmail.com	0	magnus192@hotmail.com	False	True	AAjraelOYQwti	False	147e1ce5-a516-	False
robert.mayer@integrationsoftware.se	0	robert.mayer@integrator	False	True	AFQy+VTPuv9i	False	260124da-2778-	False

Queue storage

The queue storage consists of two queues. A job queue and a version queue. The job queue contains the jobs that will be executed by the “grunt” worker role. The version queue holds the information when the plugin list was last modified or updated so that the “grunt” worker role will execute the latest plugins.

Blob storage

The blob storage consists of a container that holds all the plugins. This is where all the DLL files are stored whenever an administrator uploads or edits a plugin. The “grunt” worker role loads these plugins on startup and whenever the version queue is updated.

3.2.4 Plugins

Plugins are DLL-files which are stored in a blob storage. The primary task of these plugins is to retrieve and process data from a specific data source and create events containing this data.

These events are then sent to Integration Manager.

For example a currency tracking plugin fetches XML-formatted data from an RSS-feed on a currency tracking website. A plugin also describes in what way it can be configured. E.g. a currency plugin can describe a text-field in the GUI which asks the user to input what currencies to track.

3.2.5 Service Bus Relay

As explained earlier in Chapter [2.3.5 Azure Service Bus Relay](#), the service bus relay is used to connect to Integration Manager's WCF-service. This service bus relay is used to send events (data) from the “grunt” worker role to Integration Manager. The connection string to the service bus can be specified per configuration, i.e. a single user may have configurations which send logs to different Integration Managers, see Appendix [B3. View for creating a configuration](#).

3.2.6 Researching Azure Marketplace

In order to specify a project design we needed to know the limitations and requirements of listing an application on Azure Marketplace. According to our supervisor at Integration Software, Microsoft representatives have said and advertised that Azure Marketplace would be updated so that anything created with Azure techniques can be packaged and deployed to Azure Marketplace. This means that a customer can, through Azure Marketplace buy the application and all of its components and then host it with their Azure account. After some research and our supervisor asking Microsoft representatives about Azure Marketplace it seemed as though this was not possible as of March 2015. Figure 17 illustrates how this would have worked.

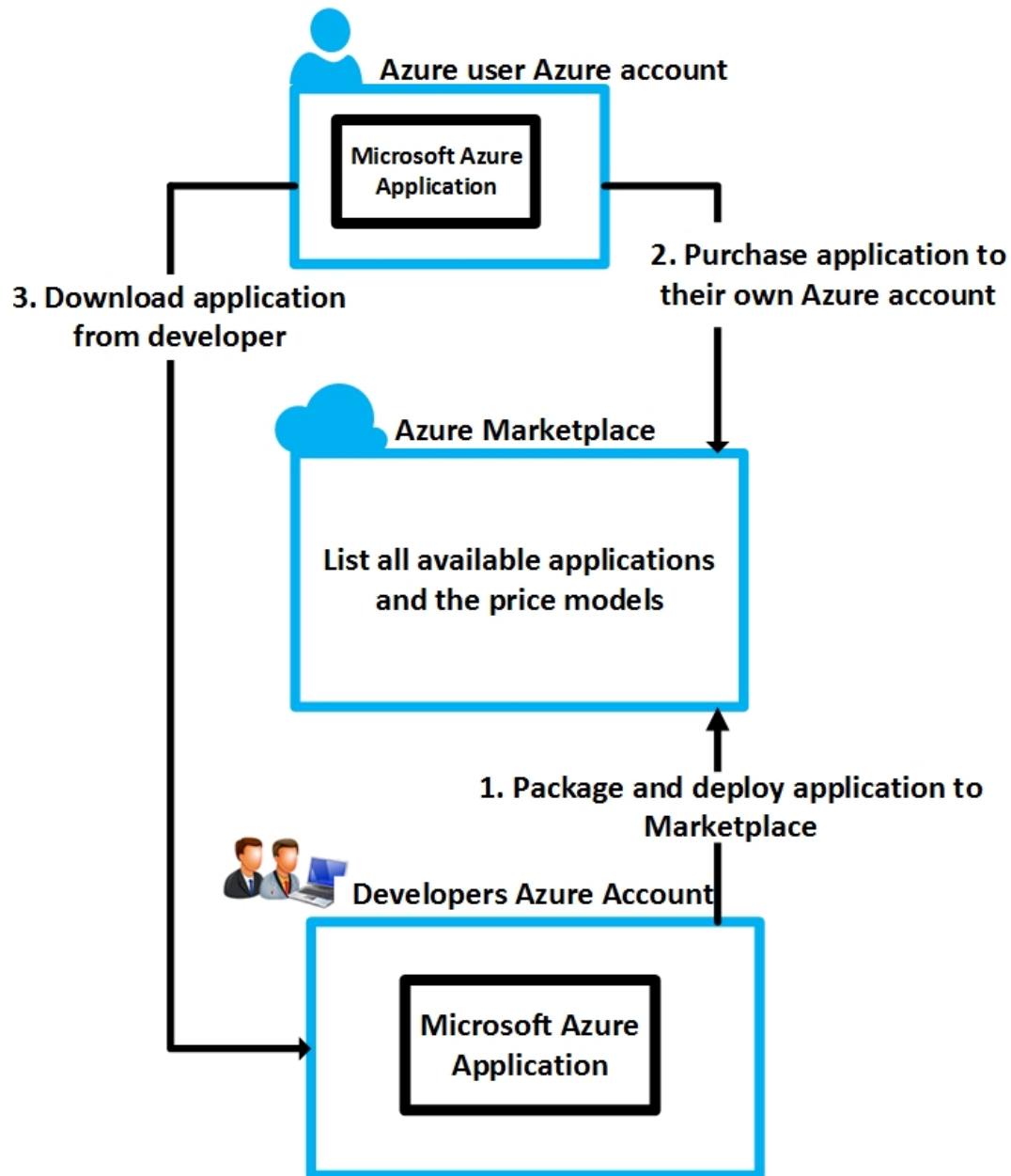


Figure 17. *How Azure Marketplace was said to work with the new updates*

Due to this situation we had to redesign accordingly. This meant that our application must be hosted by Integration Software and therefore Azure Marketplace would act as a portal to the application. Figure 18 shows how the application is accessed through Azure Marketplace.

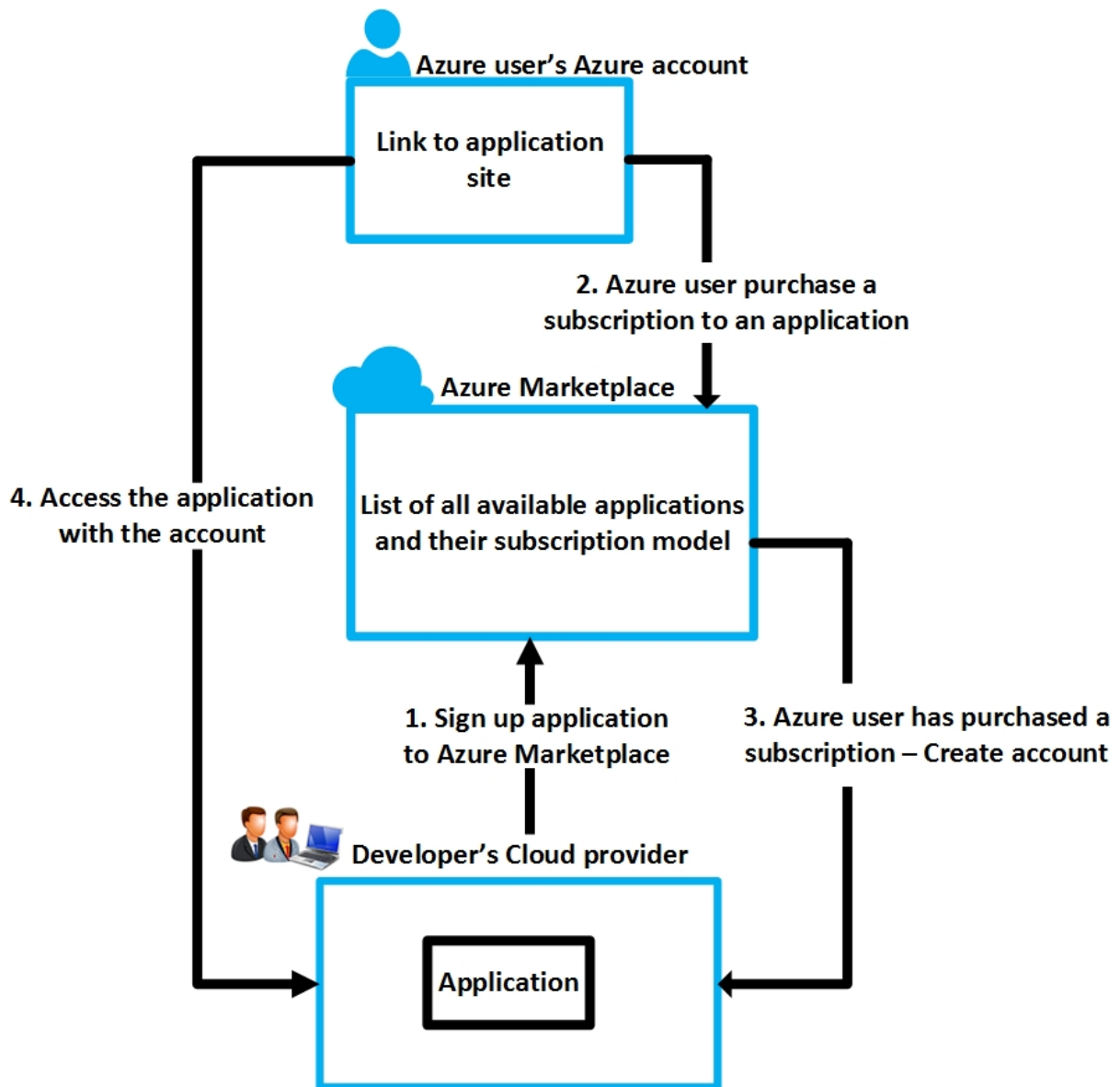


Figure 18. *How Azure Marketplace works now*

With this model (Figure 18) of Azure Marketplace, the developer needs to host the application and manage all the workload. This means a heavier workload for the administrator, since the administrator needs to manage and monitor all the data and processing power used. This Azure Marketplace model means that the application owner has to specify subscription plans. This adds complexity since a number of factors need to be considered, e.g. maintenance of the application to help determine pricing. One of the benefits of this model is that the application is not dependent on Azure Marketplace. This means that the application can be “listed” on other cloud service providers such as Amazon or Google.

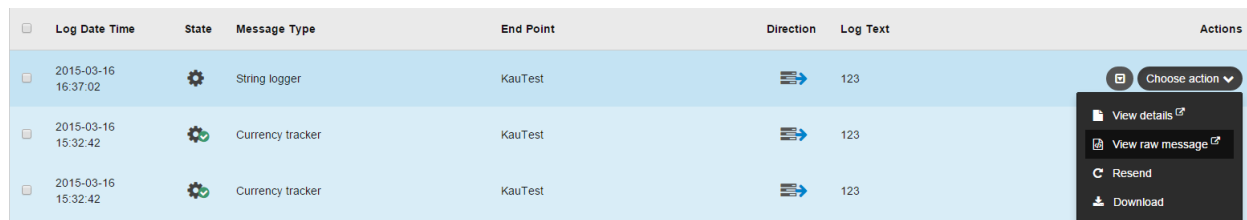
The other model (Figure 17) means that a developer can deploy the application to the Azure Marketplace and the customer may host it themselves, this is comparable to licensing of software.

The “old” way of deploying the application to Marketplace did not change the design or structure of the application at all. However, we had to implement a RESTful API to be able to communicate with the module on Azure Marketplace. Implementing the “new” way of Azure Marketplace would probably not need to implement an interface with Azure Marketplace, you could just upload the entire project and users would download the project to their Azure account.

3.3 How does the application work?

The application requested by Integration Software was a logging agent (see Chapter [1.2 Objective](#) for brief explanation). Figure [14](#) and [15](#) illustrates all the components of the application and the following steps explain how the application works with a concrete example.

1. The Azure user configures the application, i.e. what to be logged, through a web interface (the cloud service's web role) (see Appendix [B3. View for creating a configuration](#)). This configuration is then stored to an Azure Table storage called configurations (explained earlier in [3.2.3 Azure Storage](#)).
E.g. User: User1; Plugin: Currency; Interval: Every 24 hours; pluginConfig: USD.
2. The cloud service's "master" worker role gets all the configurations from the Azure Table storage and schedules the job whenever a configuration is due.
E.g. every 24 hours the "master" worker role schedules the configuration (specified in step 1) to the job queue.
3. The cloud service's "grunt" worker role polls the job queue and executes the plugin specified by the configuration.
4. With the help of the plugin, the "grunt" worker role executes and fetches raw data from external sources.
E.g. the "grunt" worker role executes the currency plugin which gets an XML file of the USD currency.
5. The cloud service's "grunt" worker role then creates log events using the data and sends it to Integration Manager through a Service Bus Relay for further handling of the data.
6. The Azure user can now see the data in form of logs (see Figure 19) through Integration Manager.
E.g. every 24 hour a currency log of USD will be shown on Integration Manager.



<input type="checkbox"/>	Log Date Time	State	Message Type	End Point	Direction	Log Text	Actions
<input type="checkbox"/>	2015-03-16 16:37:02		String logger	KauTest		123	<div><div>Choose action</div><div><div>View details</div><div>View raw message</div><div>Resend</div><div>Download</div></div></div>
<input type="checkbox"/>	2015-03-16 15:32:42		Currency tracker	KauTest		123	
<input type="checkbox"/>	2015-03-16 15:32:42		Currency tracker	KauTest		123	

Figure 19. A part of Integration Manager's user interface

3.4 Alternative design

The initial design, proposed by Integration Software (see Figure 20), included a web-API which would collect the raw data and send it back to the caller. The proposed design was composed of three major components - A Cloud Services application, which would talk to the web-API consisting of an Azure Web App and Azure API Management portal. The benefits for implementing this design were as follows:

- The web-API is very likely to change because adding an external data source could mean writing code for fetching that particular data. Running the web-API and cloud service separately would make it so that changes could be done to the web-API without affecting the cloud service.
- A web-API can handle multiple requests concurrently, which is important for good performance.
- Websites applications are scalable, which can be done automatically (explained in Chapter [2.3.8 Azure Web App](#)).
- With Azure API Management portal (explained in Chapter [2.3.6 API Management Portal](#)) it would be possible to offload several tasks, i.e. usage statistics, authentication and conversion from JSON to XML and vice versa.

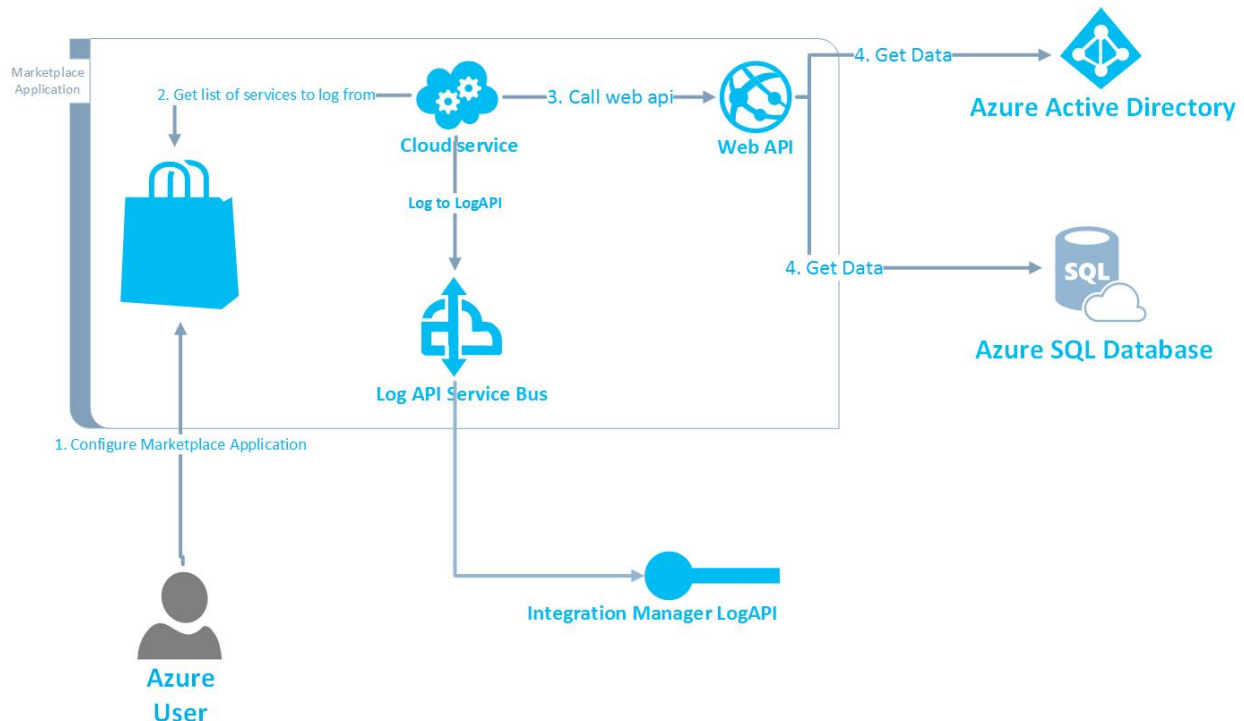


Figure 20. Initial application design proposed by Integration Software

Some of the key points for not choosing this design were as follows:

- We came up with another solution to avoid having to restart the cloud service every time another data source is added - dynamically loaded plugins, which the program loads at runtime rather than startup. This means updating or adding plugins while the application is running is not an issue.
- Having the cloud service do all the grunt work is preferable because a cloud service can scale further than a Web App (see the chapter about Cloud Services [2.3.2 Cloud Services](#) for more information about how cloud services scale).
- Having two separate components (Cloud services and Web API) would require three separate services to run, namely a Cloud service, a Websites application and an API Management service, thus yielding higher complexity and potentially higher maintenance.

3.5 Chapter summary

In this chapter we explained the preparations we made in order to gain sufficient knowledge about the different Azure services. This was necessary in order to specify the design. Then we explained the different components of the design such as their job, what they represent and how they are connected with each other. Then we briefly explained how the Azure Marketplace works and how it affected our design and following that we explained how the application works with a concrete example. Finally we explained how a design was initially proposed to us by our supervisor at Integration Manager, but after revising it and understanding the different services and techniques offered by Azure and Microsoft better, we redesigned the application by replacing the initially proposed component web API to DLL files. We explained the pros and cons of exchanging the web API with DLL files.

Chapter 4

Project implementation

In this chapter the implementation of the project is thoroughly explained. First the project's structure is explained. After that the implementation of the different components is explained, beginning with Configuration and Plugin which are the datatypes used to describe the two cornerstones of the program.

4.1 The project components and code structure

Figure 21 illustrates an overview of the different components in the project and the different modules. In the following sections the implementation of these different components and modules are explained.

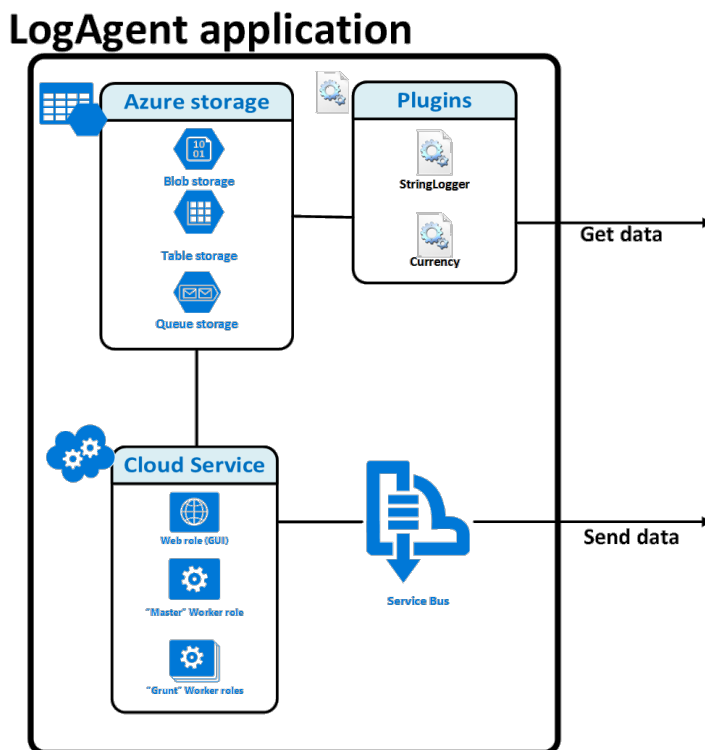


Figure 21. Finalized detailed overview of the project design and its components

Figure 22 illustrates the project structure in Visual Studio of the previously shown components and modules in Figure 21. As mentioned in chapter [3.2 Application design](#), the project consists of a web role, two plugin modules (StringLogger and Currency), two worker roles and supporting libraries. WorkerRoleMaster is the “master” worker role and WorkerRole is the “grunt” worker role. All the plugins (StringLogger and Currency) implement abstract methods from Plugin. Contracts is a library provided by Integration Software, this is used to interact with Integration Manager by using a WCF-service exposed via a service bus. CommonLibrary is code shared between several of the different roles. It contains code to handle configurations and plugins. It also contains the Configuration class. The UnitTests folder consists of all the tests for the entire project.

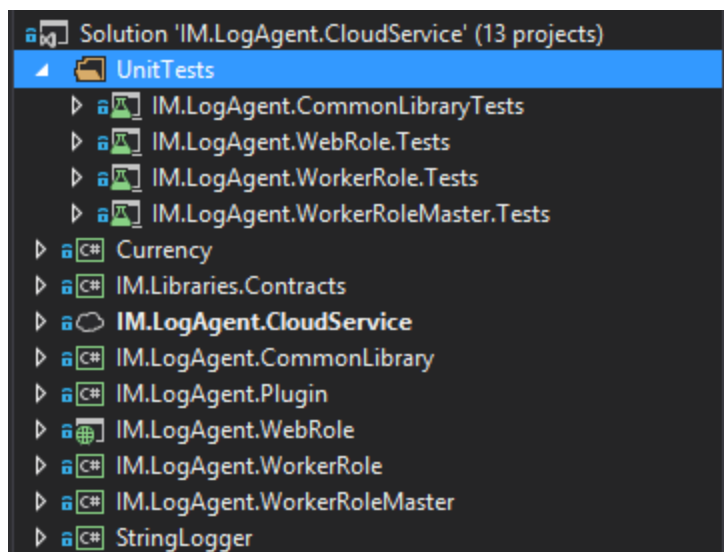


Figure 22. *Project structure in Visual Studio*

All these modules and their classes and the most crucial methods will be thoroughly explained in the following sections. Figure 23 shows all the relations between the different components in the project.

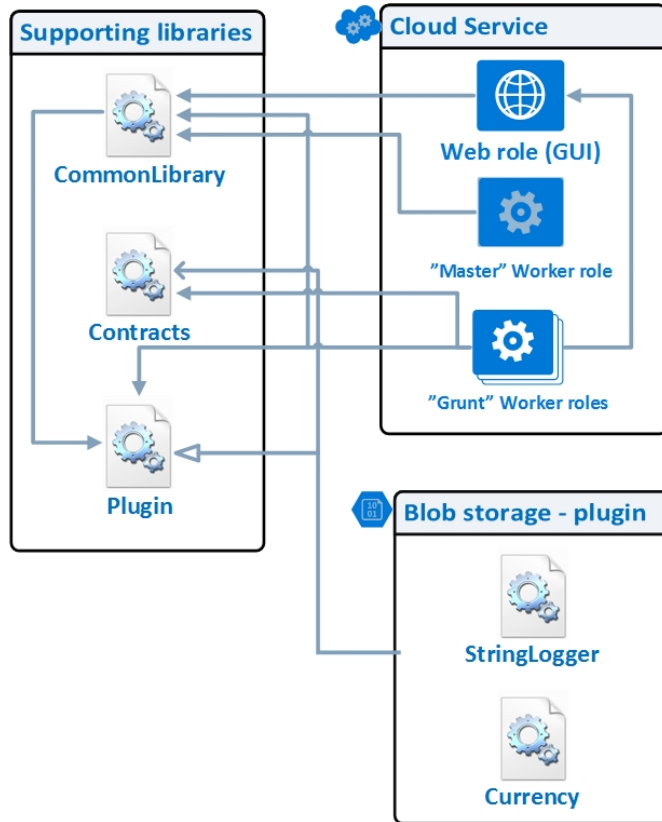


Figure 23. *The connections of the modules to each other*

4.2 CommonLibrary

The CommonLibrary is a cornerstone of the application, almost every component of the application uses it. The CommonLibrary contains crucial methods for the application such as CRUD-operations for configurations, plugins and storage. The CommonLibrary also contains two crucial data types - Subscription and Configuration.

As shown in Figure 24, the CommonLibrary consists of five classes: Configuration, ConfigurationHandler, PluginHandler, StorageHandler and Subscription. These five classes will be explained in this subchapter.

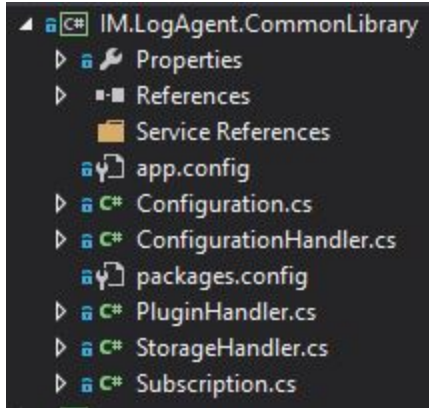


Figure 24. *The classes within CommonLibrary*

4.2.1 Configuration and ConfigurationHandler

See Chapter [3.2.1 Configurations](#) for an overview of configurations and their properties. A configuration is represented by the Configuration class. Configuration inherits TEntity, which is required to make CRUD (create, read, update, delete)-operations possible to an Azure table. The TEntity class has two important fields, namely partitionkey and rowkey which are required for saving anything to the Azure table. All other table data is mapped using reflection, looking at public properties with a getter and setter and binding that property name to the corresponding field in the table, avoiding properties with the IgnoreProperty attribute, see Appendix C [Listing 17. Configuration.cs](#) line 3-15. The ConfigurationHandler class is used for operations on the Azure Table holding configurations, such as CRUD-operations and setting up a table storage connection.

4.2.2 PluginHandler

The PluginHandler contains various methods to alter the plugin list. Some of these methods include updating, getting and uploading to the plugin list. Other methods such as serializing and deserializing dictionaries are also here, these are used to be able to store key/values in a column in the table storage. Listing 1 shows the code to serialize and deserialize a dictionary. The rest of the source code can be found in Appendix C [Listing 18. PluginHandler.cs](#)

Listing 1. *Serialized and deserialized dictionary*

```
public static string SerializeDictionary(Dictionary<string,string> dictionary)
{
    return string.Join(";", dictionary.Select(x => x.Key + "=" + x.Value).ToArray());
}

public static Dictionary<string,string> DeserializeDictionary(string dictionary)
{
    var dict = new Dictionary<string, string>();
    if(dictionary!=null)
    {
        foreach(string s in dictionary.Split(';'))
        {
            string[] keyVal=s.Split('=');
            dict.Add(keyVal[0], keyVal[1]);
        }
    }
    return dict;
}
```

4.2.3 StorageHandler

The StorageHandler class provides methods for setting up all the different Azure storages used in the application. Three different types of storage are used in the application; blobs, queues and tables. See Appendix C [Listing 19. StorageHandler.cs](#).

4.2.4 Subscription

The Subscription class is used to specify a subscription. Methods to get and set the different properties or an entire subscription reside here. A subscription separates different users in “tiers” based on a message limit restriction. Our application has four subscription tiers and each tier has a limitation on how many messages/logs they can send to Integration Manager each month. The message limit is not based on anything particular (normally it would be based on the cost of running the application), this tier-message limit is an example on how a real tier-message limit could look like. Table 3 shows the tier-message limit.

Table 3. *Subscription tier - message limit*

Subscription tier	Message limit
Free	1500
Basic	25000
Standard	100000
Premium	500000

4.3 Plugin

In this chapter we explain the implementation of the separate DLL files namely plugins (explained more in detail in Chapter [3.2.4 Plugins](#)).

A plugin is a separate DLL file represented in the code by an abstract class called Plugin. A Plugin must implement two abstract methods, see Listing 2 (explained in the next subchapter). This makes it possible for any developer to develop their own plugins with the abstract class Plugin. See Appendix C [Listing 20. Plugin.cs](#) for the source code.

Listing 2: *Plugin.cs*

```
public abstract List<Event> CreateEvents()  
  
public abstract List<string> GetConfigKeys()
```

4.3.1 Event class

The Event class is the type used to represent the log message being sent to Integration Manager. The Event class is defined in the Contracts module provided by Integration Software. The Event class has fields for information about the message, such as who is the processing user, the time it took to process the message and a key-value array where it is possible to specify any kind of information about the event.

CreateEvents creates Event objects, which represent the messages logged to Integration Manager. This is the method where the plugin fetches and processes all the data which is to be logged to IM. CreateEvents takes a dictionary containing a configuration specific to the plugin, e.g. a currency plugin has a key “currencies” which contains all the currencies the user wishes to log.

GetConfigKeys returns a list containing all the different keys which can be given a value by the user. E.g. a SQL-plugin, which logs data from a specific table, may contain the keys: host, database and table.

4.3.2 Plugin Example - Currency Plugin

The currency plugin is a simple plugin used to showcase and test all of the plugin related features in this application, see Listing 3 for the code. The currency-plugin's job is to fetch currency conversion rates from an RSS feed. Which currencies to fetch can be specified by the user in the GUI, see Figure 25. The method `GetConfigKeys` (Line 30-33 in Listing 3) supplies the GUI with the available plugin configuration that can be done, in this case "currencies". The GUI creates a textbox for that field so the user can input which currencies to fetch. This plugin configuration is then used by the plugin whenever it is called (see line 10 in Listing 3). These values are saved in the `pluginConfig` property, defined in `Plugin`.

A call to `InitializeEvents` is done before looping through the "events" property, this method initializes the "events" property with default event values, which are common to all plugins. `InitializeEvents` takes an Integer specifying how many events to initialize.

The method `GetCurrencyXML` reads the RSS feed corresponding to the currency passed in.



Figure 25. *GUI of plugin configuration a user can set*

Listing 3. Currency.cs

```
1. namespace Currency
2. {
3.     public class Currency : Plugin
4.     {
5.
6.         public override List<Event> CreateEvents()
7.         {
8.             if(pluginConfig!=null)
9.             {
10.                 List<string> currencies=new
11. List<string>(pluginConfig["currencies"].Split(','));
12.                 InitializeEvents(currencies.Count);
13.                 int i = 0;
14.                 Stopwatch sw = new Stopwatch();
15.                 foreach (var _event in this.events)
16.                 {
17.                     _event.OriginalMessageTypeName = "Currency tracker";
18.                     sw.Reset();
19.                     sw.Start();
20.                     _event.Message =
21. GetBytes(GetCurrencyXML(currencies[i++].Trim()));
22.                     sw.Stop();
23.                     _event.ProcessingTime = (int)sw.ElapsedMilliseconds;
24.                     _event.SequenceNo = 1;
25.                 }
26.                 return this.events;
27.             }
28.             return new List<Event>();
29.
30.         public override List<string> GetConfigKeys()
31.         {
32.             return new List<string>() { "currencies" };
33.         }
34.
35.         public string GetCurrencyXML(string currency)
36.         {
37.             string url = @"http://themoneyconverter.com/rss-feed/" + currency +
38. "/rss.xml";
39.             using (var client = new WebClient())
40.             {
41.                 client.Encoding = Encoding.UTF8;
42.
43.                 return client.DownloadString(url);
44.             }
45.         }
46.     }
47. }
```


4.4 The Cloud service

As mentioned earlier in Chapter [3.2.2 The Cloud service](#) and shown in Figure [15](#) and Figure [21](#), the Cloud service consists of three components; A web role which handles the frontend (GUI) and a “master and a “grunt” worker role which represent the backend. These components will be explained in the next coming subchapters.

4.4.1 Web Role

The web role is built with the MVC architectural pattern. The web role is basically an MVC 5 ASP.NET project in Visual Studio (web role is explained in Chapter [2.3.2 Cloud Services](#)). The web role consists of many files, many of which are automatically generated. Only the implementation of the most important features will be explained in the upcoming subchapters.

4.4.1.1 Administrator page

To access the administrator page, the user needs to be authorized. This is done with the authorization attribute, see Listing 4. The link to the administrator page is only visible to the authorized users by using an if-statement, this solution should be considered as a temporary fix to ensure that the authorization system works, see Listing 5. The proper way to do this would be by using roles or at the very least reading the administrator users from an Azure table, which would remove the redundancy of having to list users in several locations and make it easier to add/remove administrator users. This was not done due to lack of time.

The administrator controller has the authorize attribute which means a user has to be authorized in order to access the actions of the controller, see Listing 4 and Listing 5. The ActionLink, which directs the user to the administrator page, is only displayed if the user is authorized. Figure 26 shows the administrator menu.

Listing 4. *Controller user authorization*

```
[Authorize(Users="example@hotmail.com, mvgplz@integrationsoftware.se")]
```

Listing 5. *View user authorization*

```
@if(User.Identity.Name.Equals("example@hotmail.com"))
{
    <li>@Html.ActionLink("Admin", "Index", "Admin")</li>
}
```

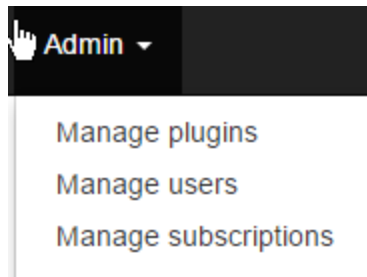


Figure 26. *Administrator menu*

As shown in Figure 26, the administrator page consists of three features; manage plugins, manage users and manage subscriptions.

4.4.1.2 Plugin management

Authorized administrator users may upload and delete plugins from the plugin management page.

The user uploads a DLL file with the input type “file” using a Html form. The data in the form is sent to the controller, see Listing 6. The Action UploadPlugin in the AdminController.cs is called, see Listing 7. This Action takes a file and calls a method in PluginHandler - UploadPlugin which uploads the file to the Azure blob storage, see Listing 8. The functionality to remove plugins was never implemented, but considered. It was of lower priority because managing plugins (editing the Azure storage) could be done with an external application called Azure Storage Explorer 6 [\[51\]](#). Figure 27 shows the GUI for plugin management

Listing 6. *Html.BeginForm*

```
@using (Html.BeginForm("UploadPlugin", "Admin", FormMethod.Post, new { @class =  
"form-horizontal", role = "form", enctype = "multipart/form-data" }))  
{  
    <div class="panel panel-primary">  
        <div class="panel-heading"><h2 class="panel-title">Upload plugin</h2></div>  
        <div class="panel-body">  
            <input type="file" name="plugin" accept=".dll"><br />  
            <input type="submit" class="btn btn-default">  
        </div>  
    </div>  
}
```

Listing 7. *ActionResult UploadPlugin in AdminController.cs*

```
public ActionResult UploadPlugin(HttpPostedFileBase plugin)
{
    if (plugin != null && plugin.ContentLength > 0)
    {
        PluginHandler.UploadPlugin(plugin.InputStream, Path.GetFileName(plugin.FileName));
    }
    // redirect back to the index action to show the form once again
    return RedirectToAction("Index");
}
```

Listing 8. *Method UploadPlugin in PluginHandler.cs*

```
public static void UploadPlugin(Stream stream, string fileName)
{
    versionQueue.AddMessage(new CloudQueueMessage("update"));
    CloudBlockBlob blob = container.GetBlockBlobReference(fileName);
    blob.UploadFromStream(stream);
}
```

Plugin management

The screenshot displays a web interface for plugin management. At the top, there is a blue header bar with the text "Upload plugin". Below this, there is a file upload section with a text input field containing "Ingen fil har valts" and a "Välj fil" button. Below the input field is a "Skicka" button. Below the upload section, there is a "Remove plugin" section. It features a dropdown menu with "Currency" selected and a blue "x" icon to its right.

Figure 27. *GUI of plugin management*

4.4.1.3 User management

Authorized administrators may monitor, register, edit and remove users from the user management page. To manage and authorize users the ASP.NET Identity framework is used. In order to use the Identity framework with Azure Table storage a NuGet package is needed - ElCamino.AspNet.Identity.AzureTable. This package replaces the code used to interact with the database from SQL server to Azure Table storage. The Web API (explained further in Chapter [4.4.1.6 Web API](#)) also uses the Identity framework to create users based on API-requests. The method to register a user is already included when creating an MVC 5 ASP.NET application, this functionality was simply moved to the user management page. Listing/Removing/Edit users was considered but never implemented due to lack of time. It was of lower priority because managing users (editing the Azure storage) could be done with an external application called Azure Storage Explorer 6 [\[51\]](#). Figure 28 shows the GUI of the user management.

User Management





Register user		List all users			
User name	Email	Message count	Subscription	Edit	Remove
"User1"	"example@mail.com"	"1337"	"PREMIUM"		
"User2"	"example2@mail.com"	"101"	"FREE"		

Figure 28. *GUI of user management*

4.4.1.4 Subscription management

The idea was that authorized admin users may edit the subscription tiers (explained in Chapter [4.2.4 Subscription](#)) from the subscription management page. This was however not implemented due to lack of time and being of lower priority. Managing subscriptions (editing the Azure storage) could already be done with an external application called Azure Storage Explorer 6 [\[51\]](#).

4.4.1.5 Third party login authentication

Implementing third party login authentication is trivial. This feature is included in MVC 5 ASP.NET, in order to use third party authentication you need a key or id and a secret key. This information is obtained through the different third parties, usually by signing your application to the third party, see Listing 9.

Listing 9. *Microsoft Account Authentication*

```
app.UseMicrosoftAccountAuthentication(  
    clientId: "Your ID",  
    clientSecret: "Your Secret");
```

4.4.1.6 Web API

The web API is used to manipulate the user data in the Azure table, such as CRUD operations. The web API uses basic access authentication, which is a way for an HTTP user agent to provide a username and password in a request, see Listing 10, this solution is not optimal from a security perspective but it was deemed sufficient. The Web API is used to interface with Azure Marketplace to automatically update users with an API call.

Listing 10. *Method for basic authentication*

```
private bool Authorize()
{
    try
    {
        var auth = this.Request.Headers.Authorization.ToString();
        return auth.Equals("secret");
    }
    catch (NullReferenceException)
    {
        return false;
    }
}
```

4.4.1.7 Configuration GUI

The configuration GUI consists of two views. The first is Index.cshtml, this view presents the user with all the information about the currently running configurations from the Azure table (see Figure 29). From this view the user can CRUD configurations.

Logging Agent

Home

About

Contact

Admin

Hello bui.michael@hotmail.com!

Log off

Configurations

Create configuration

Reset

Plugin	Interval/schedule	Created date (GMT+0)	Details	Edit	Delete
Currency	* / 30 * * ? * MON,TUE,WED,THU,FRI,SAT,SUN	2015-03-30 16:38:21			
StringLogger	* / 45 * * ? * MON,TUE,WED,THU,FRI,SAT,SUN	2015-03-23 15:17:19			

© 2015 - Michael Bui & Magnus Pedesen

Figure 29. *Index page for configurations*

The other view is a shared EditorTemplate [\[42\]](#) folder, which contains the views (see Appendix C [Listings 22. EditorTemplates](#)) for Edit.cshtml and Create.cshtml. Edit.cshtml and Create.cshtml uses the same editor template, the only difference is that they correspond to different actions in the controller. Listing 11 shows how this template is created for Create.cshtml and Figure 30 shows the GUI of the shared view.

Create configuration

Plugin

Currency ▾

pluginConfiguration[currencies]

Set interval

Common settings

Every 30 seconds ▾

Show advanced settings

Manage event

Add key-value

Key	Value	Remove
Admin	User1	✕
Secret	123	✕

LogAgentId:

EndPointUri:

Hide advanced settings

Setup Service Bus Relay

Namespace: Service path:

SAS key

Name: Connection String:

Submit

Cancel

Figure 30. Shared view to create and edit a configuration

Listing 11. Configuration create view form

```
@using (Html.BeginForm("Create", "Configurations", FormMethod.Post, new { @class =  
"form-horizontal", role = "form" }))  
{  
  
    @Html.EditorFor(m=> m)  
  
}
```

4.4.2 “Master” Worker Role

The master worker role coordinates the “grunt” worker roles by scheduling jobs in a cloud queue. It schedules configurations based on a cron schedule. A cron schedule is a terse way of specifying a schedule. It looks like this:

"0 */5 * * * ?"	- Runs every 5 minutes.
"0 30 10-13 ? * WED,FRI"	- Runs every 30 minutes between 10-13 on Wednesday and Friday

A library called Quartz.NET is used to set up timers based on a cron schedule. See Listing 12 for how to schedule jobs.

Listing 12. *Setting up a Quartz.NET job and scheduling it.*

```
IJobDetail job = JobBuilder.Create<CreateWorkItemJob>()  
    .WithIdentity(conf.id, jobGroup)  
    .Build();  
job.JobDataMap["configuration"] = conf; //pass configuration to job.  
ITrigger = TriggerBuilder.Create()  
    .WithIdentity(conf.id, triggerGroup)  
    .WithCronSchedule(conf.cronSchedule)  
    .Build();  
sched.ScheduleJob(job, trigger);
```

The job to be performed is of the type `CreateWorkItemJob`. `CreateWorkItemJob`'s job is to create a work item and add it to the cloud queue. A configuration is passed to the job, which the job serializes and sends to the job cloud queue.

The scheduler polls an Azure storage table, containing configurations, looking for added or updated configurations and schedules or reschedules them respectively. A Configuration is rescheduled after its reschedule flag is set in the table. Whenever a scheduled timer triggers, a work item is added to the job queue, waiting to be grabbed and processed by a “grunt” worker role. The process for a “master” worker role is illustrated in Figure 31.

4.4.3 “Grunt” Worker Role

The method `Run()` in the worker role runs every 100ms. It starts by getting a message from the `jobQueue` and checks if plugins are up-to-date with the call to `UpdatePlugins` in `PluginHandler`. This ensures that configurations are always processed with up-to-date plugins. Then the message from the queue is deserialized and removed from the queue. Lastly the configuration (created by deserializing the queue message) is processed.

The “grunt” worker role receives messages from the job queue and version queue. The job queue is posted to by the “master” worker role to coordinate the “grunt” worker roles. The version queue is used to keep the plugins up-to-date. Listing 13 shows what the “grunt” spends all of its time doing. This process is illustrated in Figure 31.

Listing 13. *WorkerRole.cs/Run()*

```
msg = jobQueue.GetMessage();
if (msg != null)
{
    try
    {
        PluginHandler.UpdatePlugins();
        Configuration config = DeserializeQueueMessage(msg);
        jobQueue.DeleteMessage(msg);
        ProcessConfiguration(config);
    }
    ..
    ..
}
else
{
    System.Threading.Thread.Sleep(100);
}
```


4.4.3.1 Updating plugins

Plugins are only updated when required, i.e after a plugin has been updated or added. Whenever a plugin is updated or added via the graphical interface the web role posts a timestamp to the version queue which is used to determine whether the plugins are up to date or not. If the plugins need to be updated they are simply downloaded from the Azure blob storage and loaded into memory. All the plugins are reloaded into memory whenever an update to the plugin list is required instead of reloading the necessary (newly updated/added) plugins. This does not have a significant performance impact as long as the number of plugins is low, however should the plugins increase in number and become edited frequently, this might be an issue. This solution kept the implementation of updating plugins simple. The method in Listing 14 is called to ensure that plugins are up-to-date.

Listing 14. *Method that reloads plugins*

```
public static void UpdatePlugins()
{
    var versionMsg = versionQueue.PeekMessage();
    if (reloadRequired(versionMsg))
    {
        lastPluginUpdate = versionMsg.InsertionTime;
        LoadPlugins();
    }
}
```

4.4.3.2 Processing a configuration

When processing a configuration we first check if any plugin assemblies have been loaded (assemblies represent dynamically loaded DLLs) and that the user has not exceeded his message quota (line 4 in Listing 15). Then we find the right assembly from the list of assemblies (line 10 in Listing 15). We then retrieve a type corresponding to the name of the plugin from the assembly (line 20 in Listing 15). This type is a plugin so we make an instance from it and call `CreateEvents` which yields a list of events which are sent using the `SendEvent` method (line 25-31 in Listing 15). Before sending the event we need to setup a service bus connection, this is done with the `SetupServiceBus()` method (line 24 in Listing 15). Then a call to `UpdateUserStatistics` increases the message count for the owner of the configuration being processed (line 32 in Listing 15).

See Listing 15 for the code used to process a configuration.

Listing 15. *Method, ProcessConfiguration*

```

1.     public void ProcessConfiguration(Configuration config)
2.     {
3.         //Check if any plugins exist or if user can not send (messageLimit)
4.         if (PluginHandler.pluginAssemblies.Count==0||!UserCanSend(config.userId))
5.             return;
6.         Assembly assembly;
7.         Type type;
8.         try
9.         {
10.            //find the first loaded assembly which contains the type which the
11.            configuration specifies.
12.            assembly = PluginHandler.pluginAssemblies.First(x => x.GetType(
13.                config.plugin + "." + config.plugin) != null ||
14.                x.GetType(config.plugin) != null);
15.        }
16.        catch(InvalidOperationException)
17.        {
18.            //we dont have that plugin.
19.            Trace.TraceError("Plugin : {0} was not loaded and therefore not
20.            processed.", config.plugin);
21.            return;
22.        }
23.        //Either the plugin has a namespace or not but we make sure both are
24.        acceptable.
25.        type = assembly.GetType(config.plugin + "." +
26.            config.plugin)?assembly.GetType(config.plugin);
27.        try
28.        {
29.            ILogApiService service = SetupServiceBus(config);
30.            Plugin.Plugin plugin = Activator.CreateInstance(type) as
31.            Plugin.Plugin;
32.            int i = 0;
33.            foreach(Event _event in
34.                plugin.CreateEvents(PluginHandler.DeserializePluginConfiguration(config.pluginConfigs
35.                    erialized)))
36.            {
37.                SendEvent(service, _event);
38.                i++;
39.            }
40.            UpdateUserStatistics(config,i);
41.        }
42.        catch (ArgumentNullException)
43.        {
44.            //we dont have that plugin
45.            Trace.TraceError("Could not make instance of plugin named: " +
46.                config.plugin);
47.        }
48.        catch(Exception)
49.        {
50.            Trace.TraceError("Something went wrong with plugin: "+
51.                config.plugin);
52.        }
53.    }
54. }
55.

```

4.5 Scalability

As mentioned in Chapter [2.3.2 Cloud Services](#) cloud services can scale by adding more instances of a role within the cloud service. Adding additional instances is much like adding threads or forking a process, the difference is that instances do not share memory or processing power because they run on separate virtual machines. Communication between roles is done by using queues. Queue operations provide atomic access e.g. two instances can not retrieve the same message from the queue.

This means that the “grunt” worker roles can work independently of the other “grunt” worker role instances and they work together to empty the job queue faster. Hence we can increase the performance of the application dynamically by adding more instances of the “grunt” worker role. The communication between the different worker roles and web role is illustrated in Figure 31.

The initial solution had combined the “master” and “grunt” worker role into one worker role. This solution was easier and less complex to implement but we decided to implement the system with two worker roles namely a “master” and a “grunt” worker role for increased scalability.

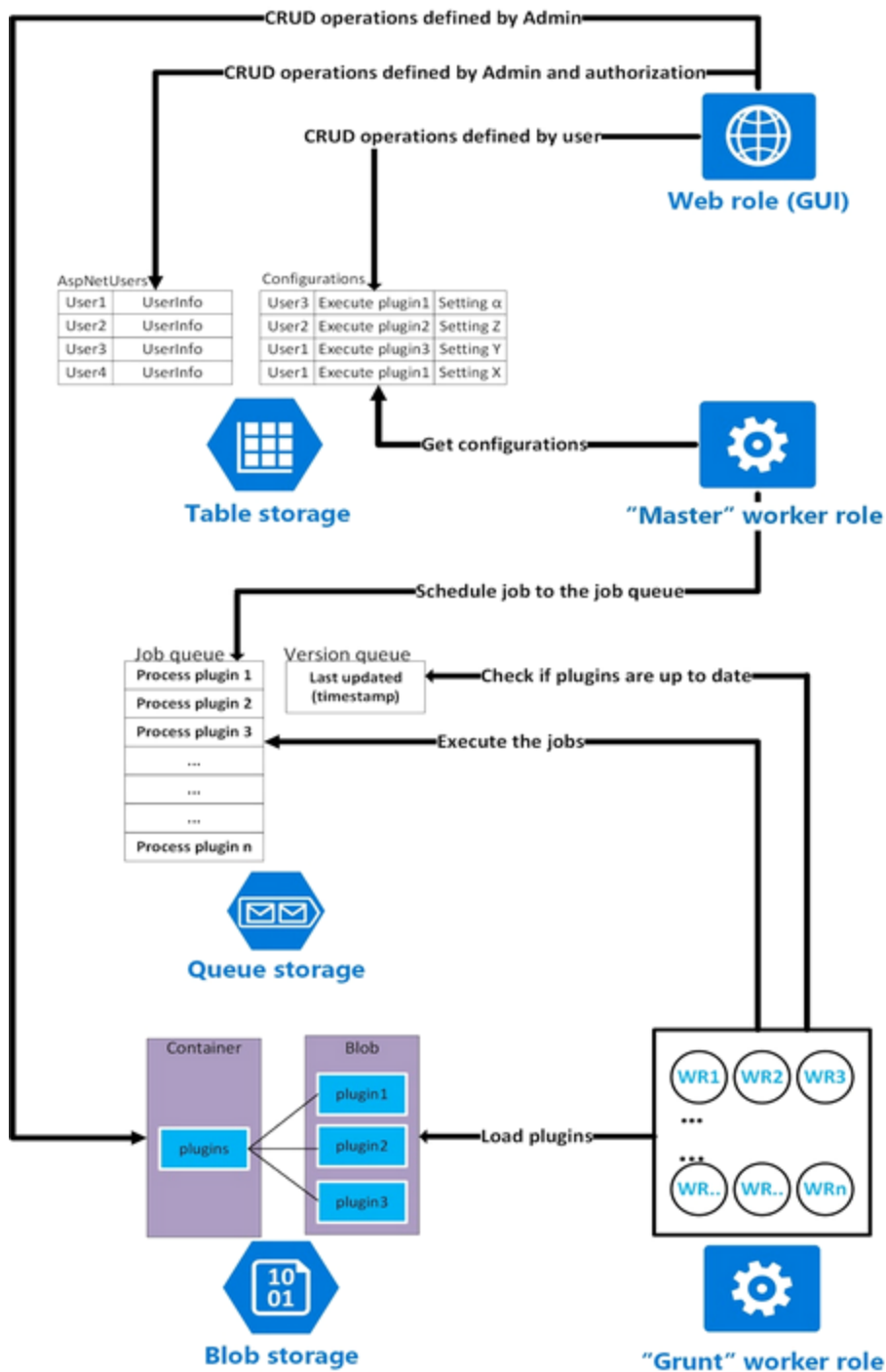


Figure 31. The interaction between the different components in the Azure storage and cloud service. The left side represents the Azure storage and the right hand side represent the cloud service.

4.6 Tests

Several parts of the project were written according to TDD (see Chapter [2.2.2 Techniques and standards](#) for an explanation of TDD). However, some parts lack automated tests as writing them was deemed to be too much work compared to testing them manually, such as GUI functionality and testing the timings of the scheduling system.

Many of the tests are used to test critical functionality such as CRUD operations to Azure tables, testing the service bus relay, loading plugins, sending events and attempting to send non-valid events.

A different way of implementing tests was introduced to us by our supervisor at Integration Software - negative tests. The principle is simple, you write a test method that you expect to fail and throw an exception. Negative tests are useful for testing if a method has exception handling among other things. See Listing 16 for an example of a negative test.

Listing 16. *Example of a negative test*

```
[TestMethod]
[ExpectedException(typeof(ArgumentNullException))]
public void TestSendNullEvent()
{
    wr.SendEvent(wr.SetupServiceBus(config), null);
}
```

The majority of the tests were deemed valuable as it would help discover bugs in the system. Although there are some disadvantages with writing tests, these are:

- Writing tests with a limited timespan for a “proof of concept” application delayed the development of the application.
- Design impacts - The design and structure of the application changed several times throughout development of the application, this caused old tests to either be discarded or rewritten.

4.7 Azure Marketplace

As mentioned in Chapter [3.2.6 Researching Azure Marketplace](#), Azure Marketplace did not work as we initially thought. Because of this we had to integrate the application with Azure Marketplace the “old” way. To integrate the application with Azure Marketplace you need to implement an interface that communicates with your application whenever a user sends a request. In the next chapter we will explain step by step how we set up a resource provider and integrated it with our application.

4.7.1 Integrating our application with Azure Marketplace

In this subchapter we will explain how we integrated the application with Azure Marketplace. The documentation for integrating an application with Azure Marketplace is slightly outdated, this is apparent when you are reading the documentation because several terms and names are outdated, such as Azure Marketplace is referred to as Azure store and “application services” is referred to as add-ons.

First we needed to register our application to the publisher site at <https://publish.windowsazure.com/>, from there you can create a draft of your Azure Marketplace application. It is very straightforward with instructions on how to, step by step, create your Azure Marketplace application. Here you can specify your subscription plans such as price, name and resource limits (called meters by Microsoft). When you publish the service, as a “staged” application (for testing purposes), you may specify which Azure subscriptions (in essence which users) may access the application, see Figure 32. However, we noticed some restrictions apply here, a BizSpark subscription (explained in [Appendix A - Installation of development environment](#)) is not eligible to purchase Marketplace applications. The reason for this is that a BizSpark subscription is a “free” subscription with credits generated by Microsoft, probably because of the possibility of having several BizSpark subscriptions and then “buying” your own Marketplace application with the BizSpark credits.

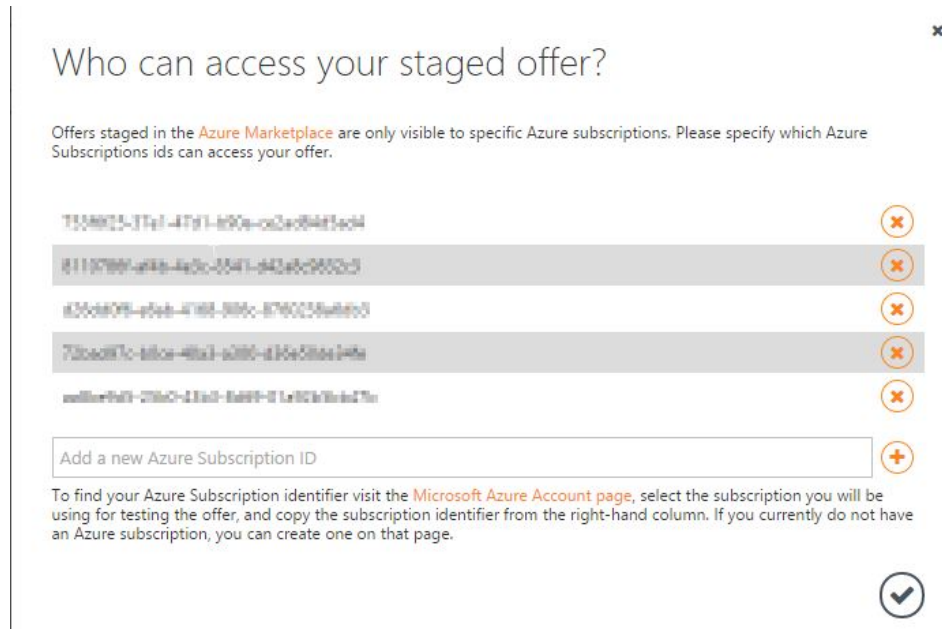


Figure 32. *Azure subscription invitations in staging*

The technical part of the integration is to implement an interface that communicates with the application e.g. when a user purchases a subscription or manages the application through Azure. To do this we needed to build a Resource Provider (RP). An RP is a web service that allows users to purchase applications from the Azure Marketplace and manage them from the Azure Management Portal. The RP has to support three different operations:

- Subscription Operations which informs the RP about the state of a user's subscription. There are four types of Subscription Operations: registered, disabled, enabled and deleted. Whenever a user purchases the application from Marketplace a registered event is sent. From this point on the RP will continue to receive these so called lifecycle events.
- Resource Operations handle five different requests that a user can perform on the application through the Azure Management Portal. Requests such as create, get, delete and upgrade resource.
- Single Sign-On Operations. The application needs to support Single Sign-On Operations (SSO Operations). This is used so the user can login from the Azure Management Portal, with a Microsoft account, to your application dashboard with the click of a button (Manage) without requiring the user to enter a username and password.

To implement the RP we started by setting up and running a sample, referenced to by the resource provider API documentation [45]. This sample RP was supplied by Metricshub on GitHub [46]. This sample implemented all of the operations and saved subscription and resource data to an SQL database. However, to get this sample running some modifications were required. We needed to deploy the RP to the cloud, since Azure Marketplace needs to be able to contact

the RP. This makes debugging more difficult as we could not launch the application in debug mode or easily print debug information. To solve this we used a cloud queue which the application posted debug messages to.

To test the RP we used a Google Chrome plugin called Advanced REST Client to send requests to the RP [\[50\]](#). These requests were equal to the requests Azure Marketplace sends.

The intent was to set up the sample RP and deploy it in order to test it with Azure Marketplace. And then from there develop an RP which would work with our application. However, we encountered issues when trying to purchase the test version of the service from Azure Marketplace. This meant we could not test the sample RP at all, since Azure would never contact the RP to begin with. The time span of the project expired and we set a stop for the deployment to Azure Marketplace. More can be read about this issue in Chapter [6.2 Problems](#).

4.8 Chapter summary

This chapter explains how the application was implemented and the attempt to integrate the application with Azure Marketplace. We present an overview of all the project components in Chapter [4.1 The project components and code structure](#). The project components, with a short description, are as follows:

- **CommonLibrary** - The CommonLibrary is shared code used by several of the components. It contains code for accessing Azure Storage and handling Plugins.
- **Plugin** - Plugin is an abstract class for implementing plugins. A sample plugin, explained more in detail in Chapter [4.3.2](#), is the currency plugin, which fetches currency data from a certain RSS feed based on what the user specifies during the setup of the configuration.
- **The cloud service** - The cloud service consists of three different roles, for more information about cloud services see Chapter [2.3.2 Cloud Services](#). The web role is the graphical user interface. The “master” worker role coordinates the “grunt” worker roles by scheduling configurations and posting data to a cloud queue. The “grunt” worker role processes a configuration by running the plugin specified by the configuration.

This chapter also mentioned how we made the application scalable by using cloud queues to communicate between the different roles and instances.

Chapter [4.6 Tests](#) explains how we used TDD to implement many of the features.

Finally we explain how we attempted to integrate the application with Azure Marketplace by creating a so called resource provider which interfaces with Azure Marketplace and our application.

Chapter 5

Results

The main goal of this project was to develop an application in the cloud platform Microsoft Azure. The other goal was to deploy the application to Microsoft's Azure Marketplace. In this chapter we will present the result of the goals discussed in Chapter [1.2 Objective](#).

5.1 Log Agent Application

The result of the application is a logging agent which consists mainly of a Cloud Services application. Surrounding parts of the application include:

- Service Bus Relay
- Plugins
- Azure storage (blob/queue/table)

The application is connected/interfaced with two other platforms, these are:

- Integration Manager. IM is used to receive user specified data from external sources. This data can then be presented to the user.
- Azure Marketplace. Azure Marketplace is used as an interface to the application. The interface calls the web-api for managing users whenever a user purchases/upgrades/terminates a subscription. More information about how we integrated the application with Azure Marketplace can be found in Chapter [5.2 Integrating the application with Azure Marketplace](#).

The base functionality of the application works as initially planned i.e. it has the functionality to fetch raw data from external sources based on a user specified interval and send that data to Integration Manager.

We managed to implement additional features that were not initially planned, some of these are:

- Ability to configure the plugins.
- A scalable cloud service.
- Ability to, more easily, extend the application by adding plugins, these plugins can be added dynamically and do not require the application to restart or recompile.
- Ability to configure the event properties (explained in Chapter [4.3.1 Event class](#)).
- Ability to configure the service bus connection info.
- Access control (only authorized users may use the application).
- Extra plugin module (StringLogger), simple plugin which sends a text specified by the user to Integration Manager.

There is however room for improvement and fine tuning of the application, such as GUI improvement, more customization and more information about how to use the application. Our supervisor at Integration Software is content with the results of the application and he stated that he is impressed by our progress of the application and the work we have done.

5.1.1 Comparison to what was planned

The initial plan was to create a logging agent and integrate it with Azure Marketplace. Much of the core design was changed during development in favor of a design involving fewer components and with scalability in mind. But the key concepts remain the same. The initial design did not take advantage of the scalable nature of a Cloud Services application. Both designs would perform the same job but the chosen design can scale in performance, which could be useful in case the workload for the application increases. The new design, unlike the one which was planned, made it possible to extend the application by dynamically adding plugins. Adding a new plugin with the original design would cause the application to be recompiled and restarted.

5.2 Integrating the application with Azure Marketplace

The secondary objective of this project was the integration of the Azure application to Azure Marketplace (explained in [1.2 Objective](#)). Our company supervisor, Robert Mayer has had numerous meetings with Microsoft's representatives regarding the new Azure Marketplace (which would make it possible to distribute applications hosted on Azure, explained in Chapter [3.2.6 Researching Azure Marketplace](#)). We started this project with the assumption that this was a possibility (to package the Azure application and deploying it to Azure Marketplace) but after extensive research and Robert Mayer contacting Microsoft's representatives regarding this matter, it was not possible as of March-May 2015. However, Robert Mayer told us that this new version of Azure Marketplace should be released by Q3 2015.

5.2.1 “Old” way vs “new” way of deployment to Azure Marketplace

Because of this we had to implement and interface the Azure application to Azure Marketplace the “old” way. The differences in how the old” and “new” way of the deployment is explained in Chapter [3.2.6 Researching Azure Marketplace](#).

In summary, the “new” way of deploying an Azure application to Azure Marketplace is much like google play or app store for mobile applications or software licenses, i.e. the developer uploads the application to Azure Marketplace and the customers can buy the application to their Azure account.

The “old” way of deploying an Azure application to Azure Marketplace is simply by interfacing your application to Azure Marketplace. This means that the developer needs to host the application and Azure Marketplace simply works as a portal to the application.

5.2.2 Conclusion

As explained in Chapter [4.7.1 Integrating our application with Azure Marketplace](#), the deployment to Azure Marketplace was never fully deployed. The reason for this is a combination of inadequate and outdated documentation. We went to a feedback meeting with Microsoft representatives in Stockholm about this matter but they would only refer us to some videos which would not answer our questions. We also tried their customer support, but we never got a good answer so we decided to stop the deployment to Azure Marketplace as we ran out of time.

5.3 An existing similar system - Logic Apps

While we were developing our log agent, Microsoft released a new feature called Logic Apps [\[44\]](#). Logic Apps is a PaaS (Platform as a service) which can be used to automate business process execution and workflow, e.g. polling a web-api for new information and notifying via email. A Logic Apps can very easily be set up using the visual designer.

This is done by using connectors (developer created separate add ons) which are a step in a chain of processes, see Figure 33.

E.g. an Azure user sets up an occurrence every minute to check whenever an RSS-feed has been updated, whenever this RSS-feed has been updated, send that data to an email and /or a database. This is a typical chain of events for Logic Apps. The possibilities with Logic Apps are quite numerous, however in this chapter, we will only explain the similarities between Logic Apps and our application.

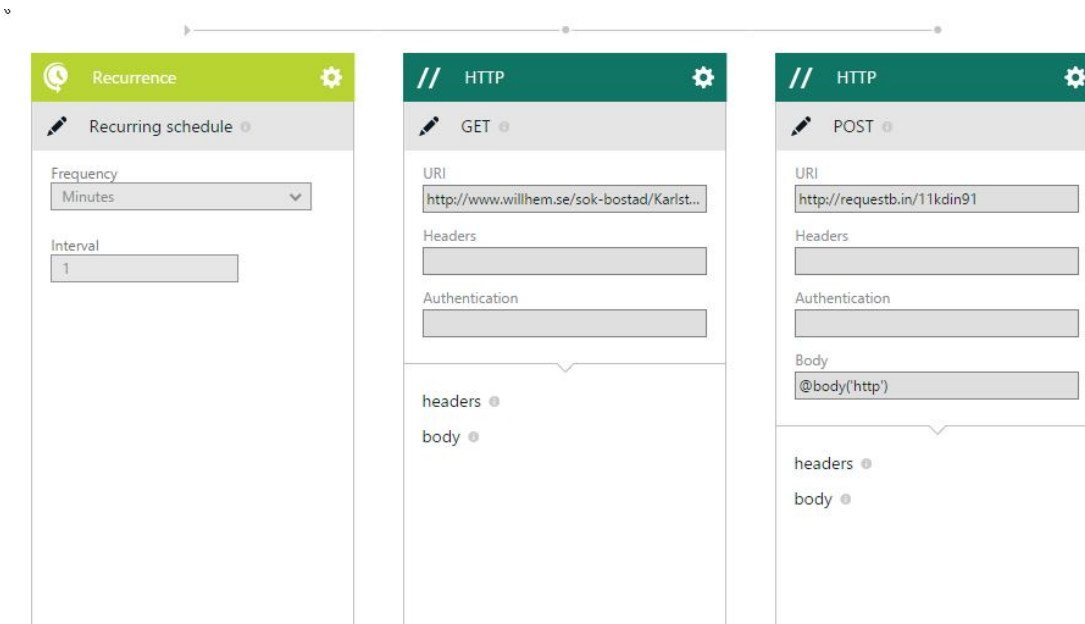


Figure 33. *Workflow for a logic app. First the recurrence is specified then two http connectors are configured, the first fetches some data from a website and posts it to another URI.*

5.3.1 Comparisons - Logic Apps vs Log Agent

Logic Apps is a similar system to what we have developed i.e. it can log on an interval to other systems. Our application is strictly tied to Integration Manager, i.e. it can only send data via an Azure Service Bus Relay to an exposed WCF-service from Integration Manager. The plugins we have developed or DLL files are comparable to a chain of connectors/events. Our application can be replaced by Logic Apps, basically all you need to do is develop a connector to Integration Manager and then you would be able to log data via this connector. What differs our application with Logic Apps is that we have a more advanced scheduler - cron scheduling. Developing plugins that send data to Integration Manager is faster and easier than developing several connectors in Logic Apps, in most cases.

5.4 Chapter summary

This chapter explained what it is we have done and accomplished. A comparison against expected results was also discussed. Then we explained why the original plan of integrating the application with Azure Marketplace did not really work as intended. And finally we introduce a similar application services developed by Microsoft, Logic Apps and compared it with our application.

Chapter 6

Conclusions and evaluation

6.1 Project summary

In this chapter we will go through the summary of the entire project. The contents of the dissertation will be divided into different sections based on the purpose and objective given in Chapter 1.1 Background and purpose and 1.2 Objective. The subjects summarized are: cloud computing, evaluating costs, application development and Azure Marketplace

Cloud computing

This project has shown us that cloud computing in Microsoft Azure is a very promising and effective platform. The reasons for this are:

- The ability to easily scale up and scale down the services and applications based on the workload.
- The applications and services provided by Azure are intuitive, easy and fast to use due to detailed documentation.
- Managing hardware such as servers and hard drives is not required as this is done by the cloud service provider.
- The data stored on cloud-platforms are more secure than on-premises due to data redundancy, security and privacy.
- After reading several sources, a cloud-based solution is cheaper and more efficient than hosting on-premises, in many cases ([\[47\]](#)[\[48\]](#)[\[49\]](#)).

These benefits of cloud computing are explained more in detail in Chapter [2.1.3 The benefits and disadvantages of cloud computing](#) and [Reliability, redundancy and privacy in the cloud](#).

Evaluating costs

One of the objectives of the project was to carefully evaluate the costs of running an Azure application. This was however not done due to lack of time and having a lower priority.

Application development

The application is a logging agent which consists of three Azure services namely an Azure Cloud Service, Azure Service Bus Relay and Azure Storage. The application uses a website as the GUI to configure and use the application. The worker roles in the cloud service act as the backend of the application. The Azure storage is our database and is used to store data such as configurations, plugins, user and job queues. Azure Service Bus Relay is used to send events to Integration Manager. Figure 34 illustrates an overview of the entire application.

LogAgent application

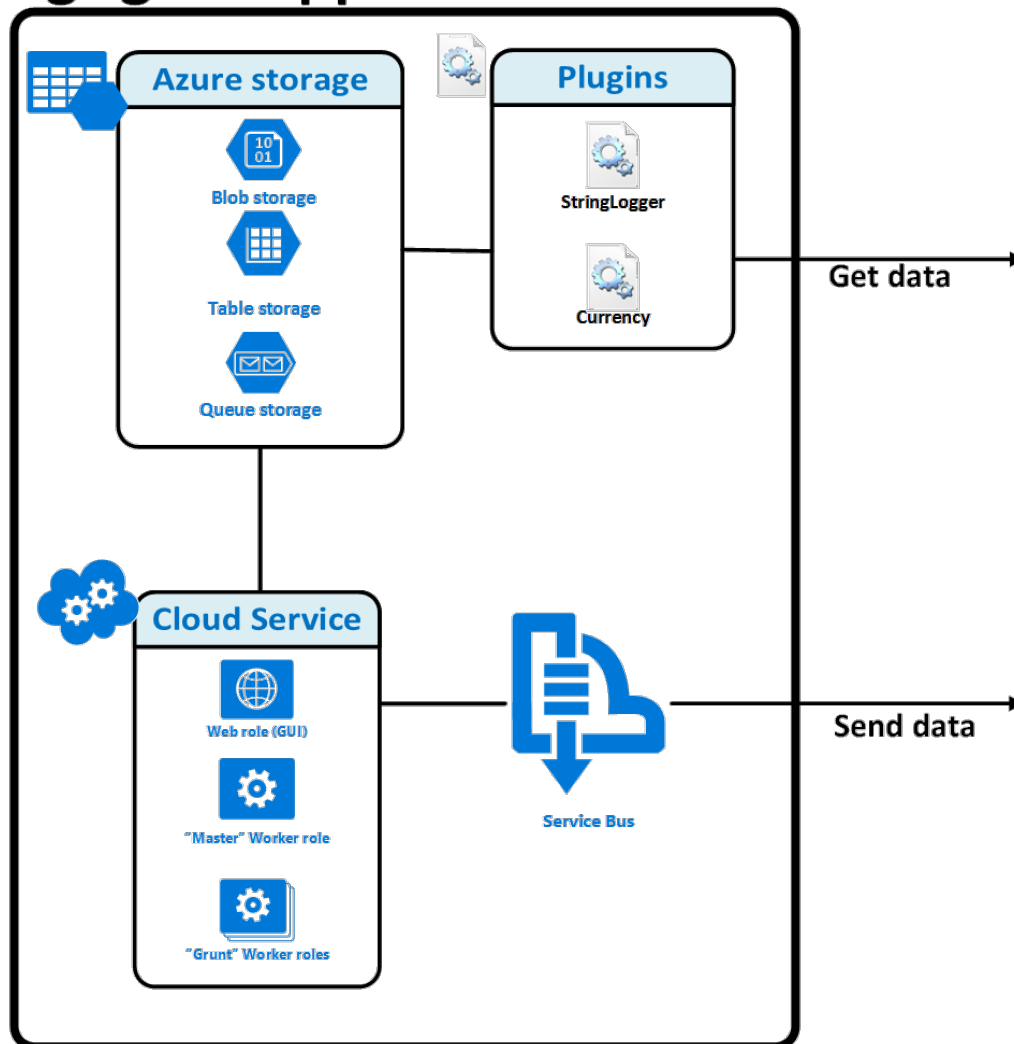


Figure 34. Detailed overview of the Log Agent application

Azure Marketplace

Packaging/deploying the Azure application to Azure Marketplace did not work as planned. We had to implement it the “old” way. This was however a valid approach and Integration Software was interested in the procedure to deploy an application the “old” way. However, as explained in Chapter [4.7.1 Integrating our application with Azure Marketplace](#) and [5.2.2 Conclusion](#) we never fully managed to deploy our Azure application to Azure Marketplace.

6.2 Problems

In this chapter we will explain the problems and other issues during this project. Smaller implementation problems will not be discussed as they were easily fixed with some searching on the Internet. Instead we will discuss the bigger picture of the problems.

Time

Limited time has certainly limited the full development of the application. Many features were considered and discussed but never implemented. The following is a list of some of these features:

- Reporting crashes in the user interface, e.g. if a plugin, for some reason, fails to load or run this should be reported to the user via the GUI.
- The ability to create plugin suites. The thought behind this was: maybe many users log very similar services, say an SQL-server, a web server and a mail server. Then maybe a suite of plugins exist for those specific logging sources. This would simplify the setup process. A suite of plugins could be similar to the software package LAMP (Linux, Apache, MySQL PHP), which provides many of the most common software for creating a web site. Suites could also be useful for organization purposes, making it possible to group plugins logging from one system. However, this idea is only useful if the premise stated earlier is true, that many users log similar services, otherwise it is unnecessary which is the reason it was not prioritized.
- Stress testing the application in order to see how effectively the application can scale as the workload increases. This would most likely require considerable work to set up properly but could be very useful for optimizing and measuring the scalability of the system. This is unfortunate because we designed the project with scalability and performance in mind.
- Not all human-computer interaction errors were covered. We implemented most of the application with the mindset that the user knows how to use the application.
- Better and more detailed user interface. This includes useful information for the user such as details, help texts and better UI design.

- Finishing the implementation of the administrator page, explained in detail in Chapter [4.5.1.1 Administrator page](#).

In conclusion, if more time to polish, test and develop the application it would become a better and more “complete” application.

Tests

We were instructed to develop the application with a TDD mindset (explained in Chapter [2.2.2 Techniques and standards](#)). This was however overlooked due to various factors such as experience, time, laziness or just forgetting. We would sometimes find ourselves finishing writing code and then ask ourselves why we did not implement this with the TDD-mindset. This would often result in tests being written after code was implemented, other times we would just forget about it until in later stages and it left us thinking “there should be tests for this”.

Integrating with Azure Marketplace

We ran into some problems when we tried to interface the application with Azure Marketplace. When purchasing our test application it would produce an error message, which we had to consult Microsoft support to interpret. These issues caused considerable delay. Without being able to test the application, the development halted as we waited for support from Microsoft to help resolve the issue. We never managed to complete the connection with Azure Marketplace and our application. Figure 35 roughly shows how far into the deployment to Azure Marketplace we got. We did however get the resource provider to work on premises.

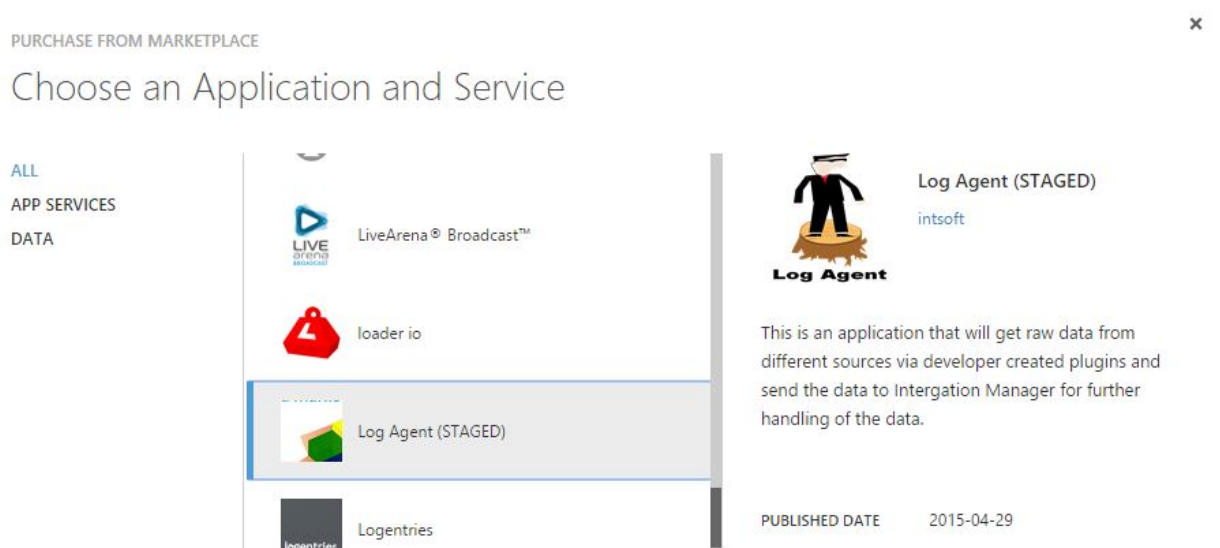


Figure 35. *Azure Marketplace, the list of all available applications (including our own staged application)*

6.3 Lessons learned

We studied several Azure techniques on a basic level during our initial phase, where we evaluated different techniques which we expected to use, and more in depth during the actual implementation. Using the cloud-platform Microsoft Azure gave us a better insight on cloud computing, how it works and what makes it useful. Many of the techniques and tools explained in Chapter [2.2 Overview of techniques and tools used](#) were new to us. This project has given us a better understanding about these techniques and tools and cloud computing in general. An interest in cloud computing has evolved in us, it is a new and fresh technique and we believe that cloud computing is the next big step in IT and we would like to work with this in the future.

In hindsight we would have organised our time differently, focusing more on the integration with Marketplace and evaluating the costs of hosting such a service. Too much time was spent fine tuning the GUI or implementing or refining some obscure feature in one of the worker roles.

6.4 Future work

The application has many areas which may be improved upon. Many features which would improve the application were not finished due to lack of time. These missing features are mentioned earlier in the chapter about problems, [6.2 Problems](#).

Since the application was not completely finished, the exact cost for the maintenance of the application was never evaluated. Knowing the maintenance cost of the application would be useful to help determine a pricing for the users of the application.

As mentioned in Chapter [5.2 Integrating the application with Azure Marketplace](#), the “new” Azure Marketplace should be released in Q3 2015. The original intention of this project was to adapt to the “new” Azure Marketplace, however this feature was not released during our time period of making the application. Possible future work for this project would be to implement the application to the “new” Azure Marketplace.

References

- [1] Various. "Cloud Computing" [Online] Available:http://en.wikipedia.org/wiki/Cloud_computing, 2015 [2015-02-10]
- [2] Microsoft. "Public preview: Microsoft Azure Marketplace" [Online] Available:<http://azure.microsoft.com/en-us/updates/public-preview-microsoft-azure-marketplace/>, 2014-10-28 [2015-02-25]
- [3] Timothy Green. "Here's why Microsoft is investing billions in the cloud" [Online] Available:www.fool.com/investing/general/2014/10/02/heres-why-microsoft-is-investing-billions-in-the-c.aspx, 2015 [2015-02-10]
- [4] Yogesh Girikumar. "If the servers holding the data for a cloud storage device crash, what will happen to the data, how will it be recovered." [Online] Available:<http://www.quora.com/If-the-servers-holding-the-data-for-a-cloud-storage-device-Dropbox-or-SkyDrive-crash-what-will-happen-to-the-data-how-will-it-be-recovered-and-if-it-is-not-recovered-can-they-be-sued-for-the-same>, 2015 [2015-03-02]
- [5] Microsoft. "Datacentre security, privacy and compliance" [Online] Available:<http://www.microsoft.com/en-gb/server-cloud/cloud-os/global-datacenters.aspx>, 2015 [2015-03-03]
- [6] Frank Bauerle. "The pros and cons of cloud computing" [Online] Available:<http://thoughtsoncloud.com/2014/05/pros-cons-cloud-computing/>, 2015 [2015-03-02]
- [7] Various. "Microsoft Visual Studio" [Online] Available:http://en.wikipedia.org/wiki/Microsoft_Visual_Studio, 2015 [2015-01-30]
- [8] Various. "Remote Desktop Protocol" [Online] Available:http://en.wikipedia.org/wiki/Remote_Desktop_Protocol, 2015 [2015-01-30]
- [9] Various. "Windows Server 2012" [Online] Available:http://en.wikipedia.org/wiki/Windows_Server_2012, 2015-02-10 [2015-03-03]
- [10] Various. "Chapter 6 Visual Studio online" [Online] Available:http://en.wikipedia.org/wiki/Microsoft_Visual_Studio, 2015 [2015-01-30]
- [11] Various. ".NET Framework" [Online] Available:http://en.wikipedia.org/wiki/.NET_Framework, 2015-02-10 [2015-02-10]

- [12] Various. "Windows Communication Foundation" [Online]
Available: http://en.wikipedia.org/wiki/Windows_Communication_Foundation , 2015 [2015-01-30]
- [13] Various. "Model-View-Controller" [Online]
Available: <http://en.wikipedia.org/wiki/Model-View-Controller> , 2015 [2015-01-30]
- [14] Various. "Test-driven development" [Online]
Available: http://en.wikipedia.org/wiki/Test-driven_development , 2015 [2015-01-30]
- [15] Various. "Microsoft Azure" [Online] Available: http://en.wikipedia.org/wiki/Microsoft_Azure , 2015 [2015-02-02]
- [16] Microsoft. "What is Azure" [Online]
Available: <http://azure.microsoft.com/en-us/overview/what-is-azure/> , 2015 [2015-02-02]
- [17] Microsoft. "Azure Regions" [Online] Available: <http://azure.microsoft.com/en-gb/regions/> , 2015 [2015-02-02]
- [18] Microsoft. "Cloud services" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/articles/fundamentals-application-models/> , 2014-10-23 [2015-02-02]
- [19] Microsoft. "Azure storage" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/articles/storage-introduction/> , 2014-12-11 [2015-02-12]
- [20] Microsoft. "How to use table storage from .NET" [Online] Available:
<http://azure.microsoft.com/sv-se/documentation/articles/storage-dotnet-how-to-use-tables/> 2015 [2015-03-04]
- [21] Microsoft. "Marketplace FAQ" [Online] Available:
<https://azure.microsoft.com/en-us/marketplace/faq/> 2015 [2015-02-03]
- [22] Microsoft. "How to use the Service Bus Relay" [Online] Available:
<http://azure.microsoft.com/sv-se/documentation/articles/service-bus-dotnet-how-to-use-relay/> 2015 [2015-03-04]
- [23] Microsoft. "What is API Management?" [Online] Available:
<http://azure.microsoft.com/en-us/documentation/articles/api-management-key-concepts/> 2015 [2015-03-04]
- [24] Microsoft. "SQL Database" [Online] Available:
<http://azure.microsoft.com/en-gb/services/sql-database/> 2015 [2015-03-04]

- [25] Microsoft. "Microsoft Azure Web App" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/services/websites/> 2015 [2015-02-02]
- [26] Microsoft. "About Azure Storage Accounts" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/articles/storage-create-storage-account/> 2015
[2015-03-05]
- [27] Microsoft. "Wikipedia Microsoft Azure privacy" [Online] Available:
http://en.wikipedia.org/wiki/Microsoft_Azure#Privacy 2015 [2015-03-17]
- [28] Microsoft. "Microsoft Azure trust center" [Online] Available:
<http://azure.microsoft.com/en-us/support/trust-center/> 2014 [2015-03-17]
- [29] Microsoft. "Cloud Services documentation" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/services/cloud-services/> 2015 [2015-03-17]
- [30] Microsoft. "Service Bus documentation" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/services/service-bus/> 2015 [2015-03-17]
- [31] Microsoft. "Storage documentation" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/services/service-bus/> 2015 [2015-03-17]
- [32] Microsoft. "SQL Database documentation" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/services/sql-database/> 2015 [2015-03-17]
- [33] Microsoft. "API Management documentation" [Online] Available:
<http://azure.microsoft.com/en-gb/documentation/services/api-management/> 2015 [2015-03-17]
- [33] Microsoft. "Connecting to another computer using Remote Desktop Connection" [Online]
Available:
<http://windows.microsoft.com/en-gb/windows/connect-using-remote-desktop-connection#connect-using-remote-desktop-connection=windows-7> 2015 [2015-03-17]
- [34] Microsoft. "Visual Studio Community 2013" [Online] Available:
<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx> 2015
[2015-03-17]
- [35] Microsoft. "Visual Studio Online Pricing" [Online] Available:
<https://www.visualstudio.com/en-us/products/visual-studio-online-pricing-vs.aspx> 2015
[2015-03-17]
- [36] Microsoft. "Connect to Visual Studio Online" [Online] Available:
<https://www.visualstudio.com/en-us/get-started/connect-to-visual-studio-online-vs.aspx> 2015
[2015-03-17]

- [37] Microsoft. "Free one-month trial" [Online] Available:
<http://azure.microsoft.com/en-gb/pricing/free-trial/> 2015 [2015-03-17]
- [38] Quartz .NET. "CronTrigger" [Online] Available:
<http://www.quartz-scheduler.net/documentation/quartz-2.x/tutorial/crontriggers.html> 2015 [2015-03-29]
- [39] Integration Software. "Integration Manager" [Online] Available:
<http://www.integrationsoftware.se/products/integration-manager/> [2015-03-31]
- [40] Various. "Representational state transfer" [Online] Available:
http://en.wikipedia.org/wiki/Representational_state_transfer 2015-03-25 [2015-04-07]
- [41] Various. "Server Message Block" [Online] Available:
http://en.wikipedia.org/wiki/Server_Message_Block 2015-04-09 [2015-04-06]
- [42] Bipin Joshi. "Server Message Block" [Online] Available:
http://www.codeguru.com/csharp/.net/net_asp/mvc/using-display-templates-and-editor-templates-in-asp.net-mvc.htm 2014-08-06 [2015-04-15]
- [43] Integration Software. "Integration Manager" [Online] Available:
<http://www.integrationsoftware.se/products/integration-manager/> 2014 [2015-04-29]
- [44] Microsoft. "Logic App Service" [Online] Available:
<http://azure.microsoft.com/en-gb/services/app-service/logic/> 2015 [2015-05-04]
- [45] Microsoft. "Azure Marketplace documentation" [Online] Available:
<https://github.com/Azure/azure-resource-provider-sdk> 2015-01-30 [2015-05-04]
- [46] Microsoft. "Resource provider for Azure Store" [Online] Available:
<https://github.com/MetricsHub/AzureStoreRP> 2013-06-20 [2015-05-04]

[47] Vaidy Iyer. "On-premise vs. cloud for enterprise mobility: The pros, cons and cost" [Online] Available: <http://www.appsfreedom.com/premise-vs-cloud-enterprise-mobility-pros-cons-cost/> 2014-12-22 [2015-05-05]

[48] Derrick Wlodarz. "Comparing cloud vs on-premise? Six hidden costs people always forget about" [Online] Available: <http://betanews.com/2013/11/04/comparing-cloud-vs-on-premise-six-hidden-costs-people-always-forget-about/> 2014 [2015-05-05]

[49] Herve Roggero. "Sample pricing comparison: On-premise vs. Private Hosting vs cloud computing" [Online] Available: <http://geekswithblogs.net/hroggero/archive/2013/02/25/sample-pricing-comparison-on-premise-vs.-private-hosting-vs.-cloud-computing.aspx> 2013-02-25 [2015-05-05]

[50] Various. "Advanced Rest Client" [Online] Available: <http://chromerestclient.appspot.com/> [2015-05-11]

[51] Various. "Azure Storage Explorer" [Online] Available: <https://azurestorageexplorer.codeplex.com/> 2014-08-15 [2015-05-13]

[52] Various. "Git (software)" [Online] Available: [http://en.wikipedia.org/wiki/Git_\(software\)](http://en.wikipedia.org/wiki/Git_(software)) 2015-05-18 [2015-05-19]

Appendix A - Installation of development environment

Figure [13](#) illustrates the entire project system and in this appendix the installation and setup of the different systems are explained. The entire development environment was set up by Integration Software.

In order to access the virtual machine (on-premises, which Integration Software hosts) a connection with RDP is used. RDP should be preinstalled on Windows (see [\[34\]](#) for a pictorial step-by-step in how to connect to another computer).

The virtual machine is running the operating system Windows Server 2012 R2, the operating system was installed by using an image disk containing the operating system. Visual Studio 2013 community edition was installed on the virtual machine.

In Visual Studio 2013, the .NET framework and a compiler for the programming language C# are included. The Azure SDK is pre-installed on Visual Studio 2012 or newer versions. This software can, free of charge, be downloaded and installed from Visual Studio's website [\[34\]](#).

A Microsoft account is used to access all of Microsoft's services such as Visual Studio Online and Microsoft Azure. A Visual Studio Online subscription can be purchased and set up on their website. [\[35\]](#).

There are several subscription tiers (including a free) for Visual Studio Online, basically the more you pay the more features are included. In this project the Basic subscription tier was used.

In order to connect Visual Studio 2013 to Visual Studio Online you need to add a reference to the Git repository to your Visual Studio project. In this project Git [\[52\]](#) was used. In the following reference a pictorial step-by-step instruction shows how to connect Visual Studio with Visual Studio Online [\[36\]](#).

Integration Software has a gold-partner membership with Microsoft, this includes the benefits of a BizSpark subscription on Microsoft Azure. BizSpark members automatically receive 150\$ monthly credits, lower Azure rates and no additional charge for using Microsoft's software on Azure. This subscription was convenient for testing the different Azure services and applications as it is not free to use them. However, new Azure users as of March 2015 may sign up for a free month of 200\$ credits to spend on all Azure services and applications [\[37\]](#).

Appendix B - GUI

In this appendix screenshots of the front-end (GUI) of the application are shown.

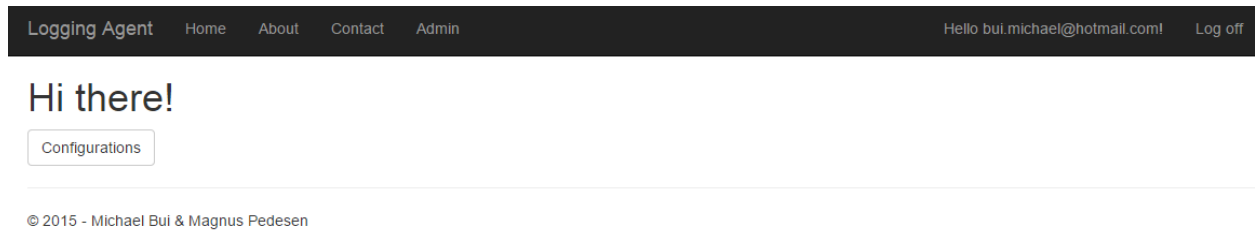


Figure 36. *Index page*

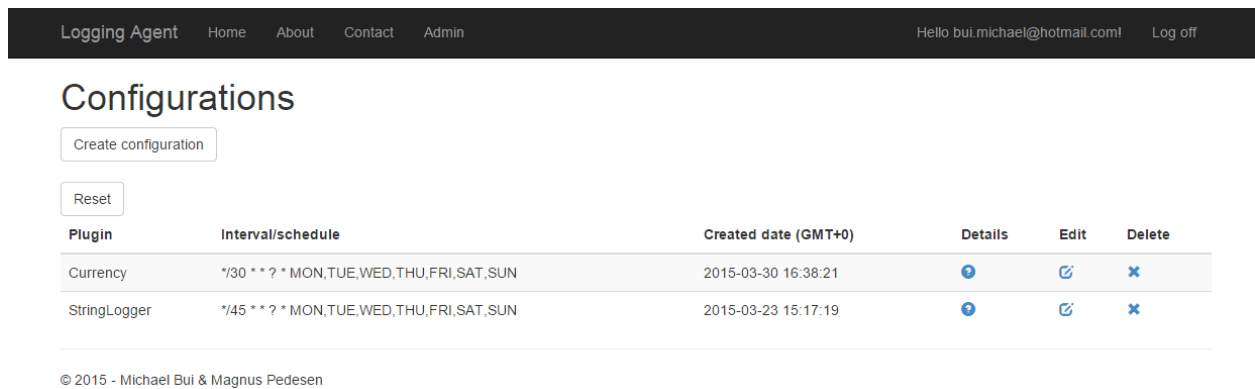


Figure 37. *View to monitor created configurations*

Create configuration

Plugin

Currency ▼

pluginConfiguration[currencies]

Set interval

Common settings

Every 30 seconds ▼

Show advanced settings

Manage event

Add key-value

Key	Value	Remove
Admin	User1	✕
Secret	123	✕

LogAgentId:

EndPointUri:

Hide advanced settings

Setup Service Bus Relay

Namespace: Service path:

SAS key

Name: Connection String:

Submit

Cancel

Figure 38. View for creating a configuration, the edit view uses the same view

Plugin management

Upload plugin

Välj fil Ingen fil har valts

Skicka

Remove plugin

Currency ▼ ✕

© 2015 - Michael Bui & Magnus Pedesen

Figure 39. Administrator page/plugin management

User Management

[Register user](#) [List all users](#)

Create a new account.

Email

Password

Confirm password

Register

User name	Email	Message count	Subscription	Edit	Remove
"User1"	"example@mail.com"	"1337"	"PREMIUM"	✎	✕
"User2"	"example2@mail.com"	"101"	"FREE"	✎	✕

© 2015 - Michael Bui & Magnus Pedesen

Figure 40. *Administrator page/user management*

[Logging Agent](#) [Home](#) [About](#) [Contact](#) [Log in](#)

Log in.

Use a local account to log in.

Email

Password

☐ Remember me?

Log in

Use another service to log in.

Google

Facebook

Twitter

Microsoft

© 2015 - Michael Bui & Magnus Pedesen

Figure 41. *Login page, with the additional services to login with*

Appendix C - Source code

In this appendix the most important classes and methods of the application is presented.

Listing 17. Configuration.cs

```
1.     public class Configuration : TableEntity
2.     {
3.         public string plugin { get; set; }
4.         [IgnoreProperty]
5.         public string id { get { return this.RowKey; } set { this.RowKey = value; } }
6.         [IgnoreProperty]
7.         public string userId { get { return this.PartitionKey; } set {
            this.PartitionKey = value; } }
8.         public string cronSchedule { get; set; }
9.         public bool reschedule { get; set; }
10.        public string sbNamespace = "kautest";
11.        public string sbName = "RootManageSharedAccessKey";
12.        public string sbKey = "THISISASECRET!!!";
13.        public string sbServicePath = "logapi";
14.        public string pluginConfigSerialized { get; set; }
15.        public DateTime createdAt { get; set; }
16.
17.        public Configuration()
18.        {
19.            this.RowKey = Guid.NewGuid().ToString();
20.            this.PartitionKey = "none";
21.        }
22.
23.        public Configuration(string plugin, string cronSchedule, string userId =
            "none")
24.        {
25.            this.RowKey = Guid.NewGuid().ToString();
26.            this.plugin = plugin;
27.            this.PartitionKey = userId;
28.            this.cronSchedule = cronSchedule;
29.        }
30.
31.        public override string ToString()
32.        {
33.            return string.Format("plugin: {0}\\t interval: {1}\\t", plugin.GetType(),
            cronSchedule);
34.        }
35.
36.        public override bool Equals(object obj)
37.        {
38.            var conf = (Configuration)obj;
```

```
39.         return this.id.Equals(conf.id);
40.     }
41.
42.     public override int GetHashCode()
43.     {
44.         return this.id.GetHashCode();
45.     }
46. }
```

Listing 18. PluginHandler.cs

```

1.  public static class PluginHandler
2.  {
3.      public static List<Assembly> pluginAssemblies = new List<Assembly>();
4.      private static DateTimeOffset? lastPluginUpdate;
5.      private static CloudBlobContainer container =
StorageHandler.SetupBlobContainer();
6.      private static CloudQueue versionQueue =
StorageHandler.SetupCloudVersionQueue();
7.
8.
9.      //reloads plugins, if updates has occurred.
10.     public static void UpdatePlugins()
11.     {
12.         var versionMsg = versionQueue.PeekMessage();
13.         if (reloadRequired(versionMsg))
14.         {
15.             lastPluginUpdate = versionMsg.InsertionTime;
16.             LoadPlugins();
17.         }
18.     }
19.     public static void UploadPlugin(Stream stream, string fileName)
20.     {
21.         versionQueue.AddMessage(new CloudQueueMessage("update"));
22.         CloudBlockBlob blob = container.GetBlockBlobReference(fileName);
23.         blob.UploadFromStream(stream);
24.     }
25.
26.     public static bool reloadRequired(CloudQueueMessage msg)
27.     {
28.         if (msg == null || msg.InsertionTime == null) //if there is no msg we are
up to date
29.         {
30.             return false;
31.         }
32.         else if (lastPluginUpdate == null)
33.         {
34.             return true;
35.         }
36.         else if (msg.InsertionTime > lastPluginUpdate)
37.         {
38.             return true;
39.         }
40.         else
41.         {
42.             return false;

```



```

43.     }
44. }
45.
46.     public static List<string> PluginList()
47.     {
48.         return container.ListBlobs().Select((blob) =>
49.             ((CloudBlockBlob)blob).Name.Remove(((CloudBlockBlob)blob).Name.IndexOf('.')).ToList(
50.             ));
51.     }
52.
53.     public static List<string> GetPluginConfigKeys(string pluginName)
54.     {
55.         if (pluginName == null) return null;
56.         try
57.         {
58.             CloudBlockBlob blob = (CloudBlockBlob)container.ListBlobs().First(a
59.             =>
60.             ((CloudBlockBlob)a).Name.Remove(((CloudBlockBlob)a).Name.IndexOf('.')).Equals(pluginN
61.             ame));
62.             Assembly assembly = Assembly.Load(DownloadBlob(blob));
63.             Type type = assembly.GetType(pluginName + "." + pluginName) ??
64.             assembly.GetType(pluginName);
65.             Plugin.Plugin plugin = Activator.CreateInstance(type) as
66.             Plugin.Plugin;
67.             return plugin.GetConfigKeys();
68.         }
69.         catch(InvalidOperationException)
70.         {
71.             return null;
72.         }
73.     }
74.
75.     public static string SerializeDictionary(Dictionary<string,string> dictionary)
76.     {
77.         return string.Join(";", dictionary.Select(x => x.Key + "=" +
78.         x.Value).ToArray());
79.     }
80.
81.     public static Dictionary<string,string> DeserializeDictionary(string
82.     dictionary)
83.     {
84.         var dict = new Dictionary<string, string>();
85.         if(dictionary!=null)
86.         {
87.             foreach(string s in dictionary.Split(';'))
88.             {

```

```

81.         string[] keyVal=s.Split('=');
82.         dict.Add(keyVal[0], keyVal[1]);
83.     }
84. }
85.     return dict;
86. }
87.
88.
89.
90.     private static byte[] DownloadBlob(CloudBlockBlob blob)
91.     {
92.         blob.FetchAttributes();
93.         byte[] byteArray = new byte[blob.Properties.Length];
94.         blob.DownloadToByteArray(byteArray, 0);
95.
96.         return byteArray;
97.     }
98.
99.     public static void LoadPlugins()
100.    {
101.
102.        lastPluginUpdate = DateTime.UtcNow;
103.        pluginAssemblies.Clear();
104.        foreach (CloudBlockBlob blob in container.ListBlobs().ToList())
105.        {
106.            try
107.            {
108.                pluginAssemblies.Add(Assembly.Load(DownloadBlob(blob)));
109.            }
110.            catch (Exception)
111.            {
112.                Trace.TraceError("Failed to load plugin: " + blob.Name);
113.            }
114.        }
115.        if (pluginAssemblies.Count == 0)
116.        {
117.            Trace.TraceError("Did not load any plugins.");
118.        }
119.    }
120.
121. }

```

Listing 19. StorageHandler.cs

```

public static class StorageHandler
{
    private static StorageCredentials creds = new StorageCredentials("logagent",
"secret");
    private static CloudStorageAccount account = new CloudStorageAccount(creds, false);
    private static CloudQueueClient queueClient = account.CreateCloudQueueClient();

    public static CloudQueue SetupCloudJobQueue()
    {
        // Retrieve a reference to a queue. queue name must be all lowercase letters
        var jobQueue = queueClient.GetQueueReference("jobqueue");
        jobQueue.CreateIfNotExists();
        return jobQueue;
    }

    public static CloudQueue SetupCloudVersionQueue()
    {
        var versionQueue = queueClient.GetQueueReference("versionqueue");
        versionQueue.CreateIfNotExists();
        return versionQueue;
    }

    public static CloudBlobContainer SetupBlobContainer()
    {
        CloudBlobClient client = account.CreateCloudBlobClient();
        var pluginContainer = client.GetContainerReference("plugins");
        pluginContainer.CreateIfNotExists();
        pluginContainer.SetPermissions(new BlobContainerPermissions()
        {
            PublicAccess = BlobContainerPublicAccessType.Container
        });
        return pluginContainer;
    }

    public static CloudTable SetupSubscriptionTable()
    {
        CloudTable table =
account.CreateCloudTableClient().GetTableReference("Subscriptions");
        table.CreateIfNotExists();
        return table;
    }

    public static byte[] DownloadCertificate()
    {
        CloudBlobClient client = account.CreateCloudBlobClient();
        CloudBlobContainer container = client.GetContainerReference("certificate");
        CloudBlockBlob blob = container.GetBlockBlobReference("AzureStoreLatest.cer");
        blob.FetchAttributes();
        byte[] res = new byte[blob.Properties.Length];
        blob.DownloadToByteArray(res, 0);
        return res;
    }
}

```

Listing 20. Plugin.cs

```

1. public abstract class Plugin
2. {
3.     public List<Event> events = new List<Event>();
4.     public Dictionary<string, string> pluginConfig = new Dictionary<string,
5.         string>();
6.     public string name { get; set; }
7.
8.     /// <summary>
9.     /// Initializes events with default information.
10.    /// </summary>
11.    /// <param name="typeName">the type name of the plugin.</param>
12.    /// <param name="eventCount">the number of events the plugin wants
13.    initialized.</param>
14.    public Plugin(string typeName,int eventCount)
15.    {
16.        this.name = typeName;
17.        this.InitializeEvents(eventCount);
18.    }
19.
20.    /// <summary>
21.    /// Initiliazes a number of events.
22.    /// </summary>
23.    /// <param name="eventCount">Number of events.</param>
24.    public void InitializeEvents(int eventCount)
25.    {
26.        for (int i = 0; i < eventCount; i++)
27.        {
28.            Event _event = new Event();
29.            _event.ProcessingModuleName = this.name;
30.            _event.LocalInterchangeId = Guid.NewGuid();
31.            _event.ServiceInstanceActivityId = Guid.NewGuid();
32.            _event.ProcessingUser = @"IBIZ\mape";
33.            _event.ProcessName = "VisualStudio2013";
34.            _event.ProcessingMachineName = "ibizbuild2012";
35.            _event.ApplicationInterchangeId = Guid.NewGuid().ToString();
36.            _event.ProcessingModuleType = "Cloud service log agent";
37.            _event.KeyValues = new Dictionary<string, string>();
38.            events.Add(_event);
39.        }
40.    }
41.
42.    /// <summary>
43.    /// No initializing of events is done.
44.    /// </summary>
45.    /// <param name="typeName">>the type name of the plugin.</param>
46.    public Plugin(string typeName)
47.    {
48.        this.name = typeName;
49.    }
50.
51.    public abstract List<Event> CreateEvents(Dictionary<string,string>
52.    pluginConfig);
53.
54.    public byte[] GetBytes(string str)
55.    {
56.        byte[] bytes = new byte[str.Length * sizeof(char)];
57.        System.Buffer.BlockCopy(str.ToCharArray(), 0, bytes, 0, bytes.Length);
58.        return bytes;
59.    }
60.

```

```

56.     public abstract List<string> GetConfigKeys();
57.     }

```

Listing 21. CreateWorkItemJob.cs

```

1.     public class CreateWorkItemJob : IJob
2.     {
3.         private static CloudQueue jobQueue = StorageHandler.SetupCloudJobQueue();
4.
5.         public void Execute(IJobExecutionContext context)
6.         {
7.             var conf = (Configuration)context.JobDetail.JobDataMap["configuration"];
8.             AddToQueue(CreateQueueMessage(conf));
9.         }
10.
11.        public CloudQueueMessage CreateQueueMessage(Configuration config)
12.        {
13.            CloudQueueMessage msg;
14.            using (StringWriter textWriter = new StringWriter())
15.            {
16.                XmlSerializer xmlSerializer = new XmlSerializer(config.GetType());
17.                xmlSerializer.Serialize(textWriter, config);
18.                msg = new CloudQueueMessage(textWriter.ToString());
19.            }
20.            return msg;
21.        }
22.        public void AddToQueue(CloudQueueMessage msg)
23.        {
24.            jobQueue.AddMessage(msg, new TimeSpan(0, 0, 5));
25.        }
26.    }

```

Listings 22. EditorTemplates

ConfigurationViewModel.cshtml

```

1. @model IM.LogAgent.WebRole.Models.ConfigurationViewModel
2. @{
3.     ViewBag.Title = "ConfigurationViewModel";
4. }
5.

```

```

6. <div class="form-group">
7.     <div class="col-md-10">
8.         <h2>Plugin</h2>
9.         @Html.DropDownListFor(model => model.plugin, new
SelectList(ViewBag.PluginList), new { @onchange =
"UpdatePluginConfiguration(this.value)" })
10.     </div>
11. </div>
12. @{
13.     ViewDataDictionary dict = new ViewDataDictionary();
14.     if (Model.plugin != null)
15.     {
16.         dict.Add("plugin", Model.plugin);
17.     }
18.     else
19.     {
20.         dict.Add("plugin", ViewBag.PluginList[0]);
21.     }
22. }
23. <div id="pluginConfig">
24.     @Html.Partial("PluginConfigurationView", Model, dict)
25. </div>
26.
27. <h2>Set interval</h2>
28. <div id="cron-schedule" style="border-style:solid; border-color: #808080;
border-width: 1px; padding:15px">
29.     @Html.EditorFor(m => m.cronSchedule)
30. </div>
31. <br />
32. <input type="button" value="Show advanced settings"
onclick="$('#advancedSettings').toggle(); $(this).val($(this).val()[0]=='H' ? 'Show
advanced settings' : 'Hide advanced settings');" class="btn btn-default" />
33. <br />
34. <br />
35. <div id="advancedSettings" style="border-style:solid; border-color: #808080;
border-width: 1px; padding:15px; display:none;">
36.     <h2>Setup Service Bus Relay</h2>
37.     <div class="form-group">
38.         <div class="col-md-10">
39.             @Html.Label("Namespace")
40.             @Html.TextBoxFor(m => m.conf.sbNamespace, new { placeholder = "namespace"
})
41.
42.             @Html.Label("Service path")
43.             @Html.TextBoxFor(m => m.conf.sbServicePath, new { placeholder =
"servicePath" })

```

```

44.     </div>
45. </div>
46. <div class="form-group">
47.     <div class="col-md-10">
48.         @Html.Label("SAS key")
49.         <br />
50.         @Html.Label("Name: ")
51.         @Html.TextBoxFor(m => m.conf.sbName, new { placeholder =
"RootManageSharedAccessKey", size = "35" })
52.
53.         @Html.Label("Connection String: ")
54.         @Html.TextBoxFor(m => m.conf.sbKey, new { placeholder = "yourKey", size =
"35" })
55.     </div>
56. </div>
57. </div>
58.
59. <div class="form-group">
60.     <br />
61.     <input type="submit" value="Submit" class="btn btn-success" />
62.
63.     @Html.ActionLink("Cancel", "Index", null, new { @class = "btn btn-danger" })
64. </div>

```

cronSchedule.cshtml

```

1. @using IM.LogAgent.WebRole.Models;
2. @model IM.LogAgent.WebRole.Models.CronSchedule
3. @{
4.     ViewBag.Title = "CronSchedule";
5. }
6.
7. <div class="form-group">
8.     <div class="col-md-10">
9.         @Html.Label("Common settings")
10.        <br />
11.        @Html.DropDownList("commonSettings", new
SelectList(IM.LogAgent.WebRole.Models.CronSchedule.commonSettings.Keys), new
{onChange="SetIntervalCommonSettings($(this).val());" })
12.    </div>
13.    <script>
14.        $("#commonSettings option").each(function(index,element)
15.        {
16.            if($(element).val()=='@Model.setting')
17.            {

```

```

18.         $(element).prop("selected", true);
19.     }
20. });
21. </script>
22. </div>
23. <input type="button" value="Show advanced settings"
    onclick="$('#advancedIntervalSettings').toggle(); $(this).val($(this).val()[0]=='H' ?
    'Show advanced settings' : 'Hide advanced settings');" class="btn btn-default" />
24. <div id="advancedIntervalSettings" style="display:none">
25.     <div class="form-group">
26.         <div class="col-md-10">
27.             <br />
28.             @Html.Label("Seconds")
29.             <br />
30.             <input type="text" name="cronschedule.seconds" value="@Model.seconds" />
31.         </div>
32.     </div>
33.     <div class="form-group">
34.         <div class="col-md-10">
35.             @Html.Label("Minutes")
36.             <br />
37.             <input type="text" name="cronschedule.minutes" value="@Model.minutes" />
38.         </div>
39.     </div>
40.     <div class="form-group">
41.         <div class="col-md-10">
42.             @Html.Label("Hours")
43.             <br />
44.             <input type="text" name="cronschedule.hours" value="@Model.hours" />
45.         </div>
46.     </div>
47.     <div class="form-group">
48.         <div class="col-md-10">
49.             @Html.Label("Days of week")
50.             <br />
51.             @for (var i = 0; i < Model.daysOfWeek.Count; i++)
52.             {
53.                 <div style="float:left; padding-left:15px">
54.                     @{bool value = @Model.daysOfWeek.Values.ToList()[i];}
55.                 </div>
56.                 <label
    style="display:block">@Model.daysOfWeek.Keys.ToList()[i]</label>
57.                 <input type="checkbox"
    onchange="$('#checkVal_@i').val($('#checkVal_@i').val() === 'True' ? 'False' :
    'True');" />

```



```
58.             @if(value){@:checked
59.                 } />
60.             <input type="hidden" name="cronSchedule.daysOfWeek[@i].Key"
value="@Model.daysOfWeek.Keys.ToList()[i]" />
61.             <input type="hidden" id="checkVal_@i"
name="cronSchedule.daysOfWeek[@i].Value"
62.                 value = "@value"/>
63.         </div>
64.     }
65.
66. </div>
67. </div>
68. </div>
```