Efficiently migrating Java/JEE prototype application to Google App Engine PaaS Cloud

prasanth prabhakaran

Submitted as part of the requirements for the degree
of MSc in Cloud Computing
at the School of Computing,
National College of Ireland
Dublin, Ireland.

August 2014

Supervisor Adriana Chis

Abstract

The thesis proposes a software methodology that helps to efficiently migrate Java/JEE applications from traditional IT Infrastructure to Google App Engine(GAE). The main aim of our software methodology is to promote greater utilization of GAE PaaS Cloud by enabling an easier and smoother migration of existing Java/JEE web applications to the GAE PaaS Cloud.

GAE is a Platform as a Service (PaaS) Cloud that allows building and deploying scalable web applications written in various programming languages such as Java, Python, PHP and Google's proprietary languguage Go. The software methodology is applicable for migrating Java/JEE based web applications that use a Relational Database Management System(RDBMS) from the traditional corporate IT networks to GAE. We consider Java/JEE application that uses an RDBMS for migration is because Java/JEE has been one of the most popular programming technology and RDBMS has been the most widely used technology for storing the application data. We consider GAE Cloud because it is one of the most mature PaaS Cloud Platform and provides standard features such as scalability, reliability, performance, security and transactional datastore to the applications deployed on GAE infrastructure.

Currently there exists a large number of Java/JEE enterprise applications using RDBMSes that run on the traditional IT infrastructures. Migrating these applications to GAE can reap the benefits of GAE platform. However migrating an application can be a challenging task as it requires the migration of the application data from the RDBMS to GAE Datastore, modification of the application code to remove the GAE unsupported JRE classes and replacement of the SQL queries in data access layer with Datastore APIs. The software methodology proposes that an application can be efficiently migrated to GAE using a Java Source Code Analyzer Tool and an automated Database Migration Tool. The Java Source Code Analyzer Tool helps to identify the usage of both GAE unsupported JRE classes in the application's Java source files and SQL queries in the Data Access Layer of the application. The automated Database Migration Tool can migrate data from the RDBMS to the GAE local Datastore in the development machine for testing purpose or to the remote Datastore on the GAE Cloud. Before migrating the application the GAE unsupported JRE classes should be removed from the application's source files, the SQL queries should be replaced with Datastore API's and the applications's data should be migrated to GAE Datastore.

We applied the software methodology to migrate our sample set of Java/JEE applications to the Google App Engine. The results were successful as the Java Source Code Analyzer Tool helped significantly to reduce the re-engineering efforts that were essential to modify the applications befored deploying to GAE Platform. The automated Database Migration Tool was able to migrate the data successfully. To conclude, we can apply the software methodology to any Java/JEE application to efficiently migrate to GAE and hence promote a better utilization of GAE PaaS Cloud.

Keywords: Google App Engine, Java/JEE, PaaS Migration, RDBMS, DataStore, Cloud Computing, Java Source Code Analyzer, Database Migration Tool, Java Compiler API's, Multithreading.

Acknowledgements

It is a pleasure to thank those who made this thesis possible.

First of all, I would like to express my sincere gratitude to my supervisor Adriana Chis for providing me with valuable guidance and constant motivation. I appreciate her for her deep knowledge in Java technologies and helping me whenever I faced obstacles during this thesis work.

A very special thanks goes Dr. Horacio González-Vélez for accepting my thesis proposal as well as preparing to do this thesis.

I must also acknowledge Keith Brittle for helping me to prepare the literature review and for teaching the referencing techniques.

Declaration

I confirm that the work contained in this MSc project report has been composed solely by myself and has not been accepted in any previous application for a degree. All sources of information have been specifically acknowledged and all verbatim extracts are distinguished by quotation marks.

Signed .... Date ....

PRASANTH PRABHAKARAN

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Cloud computing emerged as a new computing paradigm that provides the users a computational environment. The Cloud Service Provider uses the cloud service models such as Infrastructure as a Service(IaaS), Platform as a Service(PaaS) and Software as a Service(SaaS) to deliver a computational environment to the customers. In this thesis we propose an approach to efficiently migrating a Java/JEE application to Google App Engine PaaS Cloud.

In PaaS model the service provider provides a platform where the customers can create and run their applications. The most popular PaaS offerings are from Google, Microsoft and Amazon, namely Google App Engine (GAE), Microsoft Azure respectively and AWS Elastic Beanstalk. According to Shu-Qing & Jie-Bin (2010), the GAE allows the users to build and run scalable web applications on Google's Infrastructure. The GAE provides features such as URL fetch, image processing, memory cache, efficient data storage with transactional support, query API's, automatic load balancing, user authorization, security mechanisms, and local development environment. Bunch et al. (2010) mention that the GAE provides App Engine Datastore which is schemaless object storage that provides scalable storage for the web application. A web application deployed on the GAE can utilise these features as well as the App Engine Datastore to store the application data.

In this thesis we select enterprise web applications built using Java/JEE and RDBMS technologies to migrate to GAE. The Java/JEE technology has been one of the most powerful and popular programming choice since many years. According to Bhat & Jadhav (2010),the RDBMS has been the proven technology for storing data but RDBMS technology could not meet the demands of applications that require to handle large set of unstructured data or provide elastic scalability. Therefore, by migrating a Java/JEE application to GAE can benefit from utilising GAE features and can achieve application scalability.

However,Vu & Asal (2012) mention that GAE imposes several constraints that an application must meet in order to run on the GAE infrastructure. There are major challenges in successfully migrating applications to GAE. Even though, GAE allows deploying applications written in languages such as Java, Python, PHP and Go, it provides limited programming functionalities. In other words, it restricts the application to use only a subset of programming API's in a language. Therefore it is important to identify the usage of JRE classes in the application which are not supported by GAE. In addition, the application should use GAE Datastore in order to achieve application scalability. The GAE Datastore is a proprietary schemaless NoSQL Database. Bonnet et al. (2011) mention that for an existing system that uses SQL Database choose to use NoSQL system it is required to handle the Database migration process. To be more precise, the application data in the RDBMS should be moved into App Engine Datastore. Also the application's data access layer should be modified to support the create,read,update and delete(CRUD) operations on the GAE Datastore. These are the major challenges in migrating an existing Java/JEE application that uses an RDBMS to GAE platform.

The first issue of identifying the GAE unsupported JRE classes and SQL queries in the application can be solved by developing a Java Source Code Analyzer Tool. The existing tools like Google Plugin for eclipse can be used for identifying unsupported classes. However, it is observed that Google Plugin fails to identify many unsupported JRE classes making it's usage unreliable. Besides Google Plugin do not provide any hints or solutions for code refactoring to the programmer who is not familier with GAE and NoSQL Infrastructure. The thesis has came up with a tool named Java Source Code Analyzer to handle this issue. The second issue with respect to Data migration can be solved by developing an automated Database Migration Tool. Currently there is no standard tool available to perform such Data migration from the RDBMS to GAE Datastore. The GAE provides a tool named Bulkloader for exporting and importing entities to and from Datastore in CSV and XML format files(Angabini et al. (2011)). The Bulkloader tool is distributed as a part of GAE Python SDK and requires the installation of both Python runtime as well as GAE Python SDK. In addition the user has to manually export the table data from the RDBMS to CSV files which serve as the inputs to the Bulkloader. However this process involves lot of manual work and can be quite cumbersome if the application's RDBMS has many tables containing large number of records.

This research thesis discusses the issues that can comes across on migrating a Java/JEE application to GAE and came up with a software methodology in order to efficiently migrate the Java/JEE application to the GAE. The software methodology is organized as a series of sequential steps. The application along with its data are migrated as the end result. The software methodology orchestrates the use of the Java Source Code Analyzer Tool and Database Migration Tool to efficiently migrate Java/JEE applications to GAE.

1.1 Contributions

The migration of existing applications to Cloud Platforms has gathered significant attraction due to the perceived benefits that Cloud computing platforms offers such as application scalability, load balancing, reduced operational cost. Most articles discuss topics related to application migration such as feasibility of migrating applications to various cloud platforms, cost benefits achieved, challenges involved in application migration, limitations imposed by the Cloud Platforms for migrating existing application etc. However, less research has been conducted for efficiently migrating applications to the Cloud Platforms.

The main contribution of this paper is the software methodology which promotes the adoption of GAE Cloud by providing a systematic approach to migrate existing Java/-JEE applications to GAE platform. The software methodology enables organizations to move their existing Java/JEE application to GAE potentially saving cost as well as achieving improved application performance. We developed a Java Source Code Analyzer Tool and an automated Database Migration Tool to support the software methodology. The Java Source Code Analyzer Tool is developed using the advanced features provided by the Java Platform. It can scan an application's Java source files for identifying the presence of both unsupported classes JRE specified by the GAE Platform and SQL queries in the Data access layer which needs to be replaced. The automated Database Migration Tool developed can migrate huge quantities of data from any SQL Database System used by the application to GAE Datastore. The Database Migration tool relies on Java concurrency(i.e multithreading) APIs in order to make the migration faster. Currently there do not exist such tools. Both the Java Source Code Analyzer Tool and Database Migration Tool are designed to be run with minimum configuration and reduce the migration efforts required for migrating applications running on traditional IT infrastructures to GAE Cloud. Furthermore, we carry out an empirical evaluation which shows that our software methodology is successfully applied to migrate Java/JEE applications to the GAE platform.

The thesis paper is structured as follows. Chapter 2 discuss the motivation to migrate the Java/JEE application to GAE, the possible challenges that can arise on migrating to GAE Platform and proposes a software methodology. Chapter 3 briefly discuss the design specifications, functional requirements and architecture of the proposed Java Source Code Analyzer Tool and Database Migration Tool. Chapter 4 discuss more about the underlying technical implementation details of the Java Source Code Analyzer Tool and Database Migration Tool. Chapter 5 discuss the results obtained on the evaluation of the software methodology by applying the methodology to migrate a sample set of Java/JEE application to the GAE platform. Chapter 6 presents the conclusion of the research work as well as also briefly mentions about the further works that needs to be followed.

Chapter 2

Background

2.1 Introduction

Cloud Computing has emerged as one of the standard computing paradigms. The research problem that is going to be discussed for the literature review is

> Can a Java/JEE prototype application be efficiently migrated to a Google App Engine PaaS Cloud?.

Before we proceed it is necessary to have an understanding about Cloud Computing and various cloud service models as the research problem is closely related to Google App Engine, which is a popular cloud service model from Google.

Mell & Grance (2011) define Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources such as networks, servers, storage, applications and services that can be rapidly provisioned and released with minimal management effort or service provider interaction. They state that the Cloud model is composed of five essential characteristics, three service models and four deployment models. The cloud service provider relies on cloud service models to deliver its services to its customers. According to them, there are three kinds of cloud service models namely Infra structure as service (IaaS), Platform as a service (PaaS) and Software as a service (SaaS).

In the IaaS cloud service model, the subscriber can choose any operating system image to be installed in the service provider's public data center. The subscriber is then free to install any application software over the operating system. Amazon.com is a major provider of an IaaS platform. In the PaaS cloud service model, the service provider provides a computing platform which contains an operating system, program language execution environments, compilers, application development tools, databases and web servers. The PaaS model allows the developers to develop and deploy the application in the PaaS Cloud. The key providers of PaaS are Google and Microsoft. Google App Engine (GAE) and Microsoft Azure are popular PaaS cloud platforms from Google Inc. and Microsoft Inc. respectively. In the SaaS cloud service model the service providers host applications on their data centres and their subscribers can access these applications through thin clients such as browsers from their mobile phones or personal computers. In this model, the subscribers have limited rights to modify the underlying cloud infrastructures. Saleforce.com is a major provider of SaaS.

Having defined cloud computing and cloud service models, we will look into other aspects of the research problem. The Java/JEE application prototype for the research is an application written in the Java programming language and uses a relational database management system(RDBMS) for storing the application data. The acronym JEE stands for Java Enterprise Edition, which is a collection of technologies and API's for developing enterprise applications. An RDBMS is a Database management system that stores relational data in tables.

The research problem is highly important because there are many Java/JEE applications that run on the organization's traditional infrastructures; migrating these applications to GAE can bring enormous benefits to the organization by saving costs as well as leveraging the GAE infrastructure to achieve an improved application performance.

GAE is considered for the PaaS platform in this research as it has been the most popular choice of PaaS cloud Platform in the industry. Prodan & Sperk (2013) mention that GAE is one of the important and reputable PaaS that can support scalable web applications and is powerful enough to handle large traffic peaks. Srirama et al. (2012) mention that GAE supports applications written in programming languages such as Java and Python. Another important feature of GAE is that it can scale up and down the number of application instances based on the volumes of requests to the application. The GAE proprietary Datastore technology is also one of the motivating factors in choosing GAE for this thesis. According to Buyya, R., Vechhiola, C. & Selvi, S.T (2013), the GAE Datastore is designed to scale, optimized to access data quickly, supports transaction and provide optimistic concurrency control methods. The purpose of selecting a Java/JEE application to be migrated to the GAE is because Java/JEE with an RDBMS has been the popular technologies for developing enterprise applications since a long time. Vu & Asal (2012) point out that, many organizations are reluctant to migrate their applications to the PaaS cloud because there is a lack of clarity on how to migrate the applications to PaaS clouds and also caution that the application migration can be a failure if the migration is handled incorrectly. The lack of clarity is because cloud computing is relatively new and less resources are available on effective migration to the PaaS clouds.

Most of the journal articles state that migration of an application to GAE is difficult as well as challenging. The literature review discusses the difficulties and challenges stated by these journal articles and will also demonstrate how to deal with the difficulties and challenges. The literature review is organized into four main sections under the following headings: - **Programming language incompatibility issues**, **Database migration issues**, **Google App Engine limitations** and **Google App Engine Datastore limitations**. The sections are chosen for discussion because most of the journal articles consider these as the major challenges affecting the migration.

Programming language incompatibility issues discusses the challenges involved due to the difference in the version of the programming language that is supported by the GAE and the version of the programming language used to write the application. This section also discusses about the programming restrictions imposed by GAE Platform. Database migration issues discusses the challenges involved in migrating from traditional RDBMS to GAE datastore. Google App Engine limitations discusses the limitations of the GAE. Finally, Google App Engine Datastore limitations discusses the limitations of Datastore which is used to store application data in the GAE. A discussion on these sections will give an answer to the research problem in hand.

In the next section we will be discussing the Programming language incompatibility issues in more detail.

2.2 Programming language incompatibility issues

This section discusses one of the major issues that arise during the migration. The first paragraph discusses how the programming language incompatibility issues arise.

According to Vu & Asal (2012), one of the challenges that occurs during the migration of the Java application to the GAE is the programming language incompatibility issue. Vu & Asal (2012) state that PaaS Cloud supports only limited version of the Programming language. If the application is implemented in a different version which is not supported by PaaS the application needs to be modified. In addition, GAE platform imposes restrictions to increase the application security by supporting only a subset of the classes in the Java Runtime Environment(JRE). According to Google Developers (2013g), these subset of the classes that are supported by the GAE is called JRE white list of classes. If the application contains JRE classes that are not present in the JRE whitelist the application may not run on the GAE platform. The Google Developers (2013g) also points out that the JRE classes that writes to the file system of GAE should not be present in the application. They also state that there are some restrictions in using `Thread` API classes in the application as the application can create threads but these threads cannot outlive the request that created them. Further more, the usage of `java.lang.System` API's in the application will be ignored.

In this section various ways to identify the incompatibility issues will be discussed and also a few ways to overcome these issues will be demonstrated as most of the journal articles do not provide a clear picture on how to address the incompatibility issues. Incompatibility issues can be identified by using some development tools such as Eclipse and Google Plugin for Eclipse. Eclipse is a standard development tool kit for developing Java/JEE applications. Angabini et al. (2011) recommend using Google Plugin for Eclipse for its simplicity which also eases the deployment of an application to the GAE. Once the Java/JEE application is opened with Eclipse that uses Google Plugin will shows the classes of the application that are incompatible with the GAE. However it is observed that the Google Plugin has serious limitation as it identifies only a limited set of unsupported JRE classes in the application as well as it do not offer the developer any information on how to deal with the unsupported classes. In other words even if it identified the usage of unsupported JRE classes in the application it will not offer the developer any constructive suggestion on how to handle such scenarios. These shortfalls of the Google Plugin makes it less reliable. Once the incompatible classes in the application are identified the corrective measures can be taken. These measures may involve replacing unsupportive Java classes with alternative Java classes that are compatible with the GAE. The supported Java classes are published by Google Developers (2013d). Vu & Asal (2012) state that GAE doesn't support third party libraries such as Enterprise Java Beans(EJB), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), etc. EJB technology is closely associated with JNDI and RMI; it is used for running a program code within a transaction and for remotely accessing the application. A solution to the problem would be to remove these components with a data access layer that use the transaction support from the underlying GAE Datastore. Additionally, in order to ensure that the modified application will successfully run on GAE, it would be good to compile the modified application against the supported Java classes published by Google Developers (2013d).

To conclude, Programming language incompatibility issues may arise during the migration and efficiently solving these issues helps to efficiently migrate the Java/JEE application to the GAE. However identification of GAE unsupported JRE classes in the application pose a serious challenge as there are no standard tools available and even with existing tools such as Google Plugin fails to identify many of the unsupported classes there by making it less reliable. In the next section Database Migration Issues, we discuss the challenges involved in migrating from traditional RDBMS to GAE Datastore.

2.3 Database Migration Issues

In this section we will be discussing Database Migration issues that needs to be considered during the migration to GAE Platform. The Database migration involves the transition of the application from using a traditional relational database management system to the GAE Datastore as well as extracting data from the tables in the relational database management system to GAE Datastore in the cloud. The next paragraph summarizes the views regarding data migration from the traditional platform to GAE.

According to Vu & Asal (2012) and Chauhan & Babar (2012), the GAE provides a proprietary Datastore for storing application data. The GAE Datastore is schemaless and different from the traditional RDBMS. Shu-Qing & Jie-Bin (2010) agree that the GAE Datastore provides efficient storage of data and supports efficient querying, sorting and transactions but migration of data from the traditional RDBMS system to the Google Datastore can be a challenging task. Vu & Asal (2012) and Chauhan & Babar (2012) agree that there will be issues when migrating an application to the PaaS platform and the major issues arise from the database migration. Although they acknowledge that Database migration can be challenging, complex, costly and time consuming; they state that if it is done correctly it may achieve the benefits of GAE platform. In the next section we will discuss about query migration, an issue which is closely related to database migration.

Kotecha et al. (2011) discusses about the application migration from one cloud service provider to Google App Engine. According to Kotecha et al. (2011), it is necessary to perform data migration and query migration in order to move an application from one platform to another. The application code should be modified to support the database provided by the target cloud service provider. Kotecha et al. (2011) point that query migration is an important concern which needs to be addressed for migrating the application. Kotecha et al. (2011) suggest developing a tool backed by an algorithm which can translate queries written in SQL to GQL queries there by easing the migration of application to Google App Engine Platform. The acronym SQL stands for Structured Query Language, which is used to manipulate data in a relational database management system. The acronym GQL stands for Google Query Language, which is a query language to access the entities stored in GAE Datastore(Google Developers (2013j)) and the GQL grammer is similar to that of SQL(Google Developers (2013b)). The GQL can be used to retrieve entities, however, it cannot be used to modify, delete or create an entities in the Datastore(Google Developers (2013b)). According to Google Developers (2013b), the GAE also supports Data access technologies such as Java Data Objects(JDO), Java Persistence API's(JPA) as well as Google's proprietary Datastore

API's to access the GAE Datastore. Inferring from idea put forwarded by Kotecha et al. (2011), developing a tool that can take input of SQL queries and generate its equivalent GQL queries or Datastore API calls can helps in the process of application migration to Google App Engine. However, in my opinion the development of such a tool could be time consuming as the tool needs to incorporates various relational database management systems each having different implementations of SQL. However development of such a tool may be a promising solution to reduce the tedious work of replacing the SQL queries with GQL queries/Datastore API's during the migration of application from a traditional platform/cloud platform to GAE Platform. In the next section a trivial as well as a simple data migration procedure is discussed as most of the journal articles do not provide clear information on how to migrate the data from the relational database management system to the GAE Datastore.

In this section we will discuss the feasibility of Bulkloader tool to perform a database migration. The Bulkloader is a tool that is part of GAE Python SDK to export and import data as entities to and fro from GAE Datastore(Google Developers (2013f)). The primary purpose of the Bulkloader tool is for taking Data backup from GAE Datastore. However we can utilize the Bulkloader tool for Database Migration and requires the following steps. The first step is to retrieve all the data from the tables of the RDBMS of the application in the form of CSV files. The acronym CSV stands for comma separated values and a CSV file contains data delimited by a comma. The second step is to write a script in Python programming language that tells how to translate the rows related to the tables in the CSV file into GAE Datastore entities. The third step is to execute Bulkloader command that takes a CSV file as one of the inputs and Python script as another input. The Bulkloader command uses Remote API to access the GAE Datastore services and migrates the data in the CSV files to the application's Datastore in the GAE Cloud. Currently the Bulkloader tool comes only with GAE Python SDK.

To conclude, the Bulkloader tool can be used to migrate the application data stored in the RDBMS to GAE Datastore. Therefore, it potentially can solve Database Migration Issues. However, In my opinion this may not be a practical standard solution as there involves a lot of manual work such as exporting the data from the tables of the RDBMS to CSV files, installation of GAE Python SDK as the Bulkloader comes along with GAE Python SDK, writing Bulkloader python scripts to use with GAE remote API's to move the CSV data as entites in the Datastore and finally the manual execution of scripts by invoking the Bulkloader commands from the console. Using the Bulkloader to export data can be laborious if the application has many tables and have plenty of records present in the tables. In short the data migration challenge could be simplified by developing a automated Database Migration Tool that can run with minimal user intervention. Another advantage of developing the Database Migration Tool is that it can be used as a generic solution for migrating any application's data to GAE. In the next section Google App Engine Limitations, we will discuss the limitations of the GAE Platform.

2.4 Google App Engine limitations

We will be discussing two major limitations of GAE that are percieved by some journal articles namely, GAE request processing time out and restriction of uploading a file to the GAE sandbox. These two limitations are widely considered by journal articles as the major limitations of GAE. In the next section we will be discussing the request processing time out limitation.

The request processing time out feature is widely considered as a limitation of GAE by many journal articles. Vu & Asal (2012) and Buyya, R., Vechhiola, C. & Selvi, S.T (2013) agree that GAE limits the processing of all requests within 30 seconds. Vu & Asal (2012) suggest a workaround solution to this issue by creating two services to process a request that may take a longer time. One of the services executes the request and passes the request and the intermediate results to the other service for processing, if request processing is not finished before 30 seconds. If the second service could not finish the request, it will pass the intermediate results to the first service and this cycle continues until the processing of the request is completed. According to Google Developers (2013b) the request processing time out has been set to 60 seconds and GAE is optimized for applications with short-lived requests, typically those that take a few hundred milliseconds. Zahariev (2009) does not consider the time out limit as a limitation instead the time out serves to assure the stability of the system as well as conserves the resources of the system. In my opinion request processing time out is not a limitation as the web request to a resource must be served in a shorter time and should not keep the user waiting for a response. In short, if the server takes a longer period to process the request it should time out the request. The next section demonstrate how to handle the request processing time out in the application.

According to Martins (2012) GAE throws an Exception named DeadlineExceededException if the processing time of the request exceeds more than 60 seconds. Martins (2012) outlines some possible scenarios that are the root causes for DeadlineExceededException and suggests some corrective measures to follow in order to eliminate it. In my opinion, it is also important to consider the optimization of the components of the application so that the application can serve the request faster within the time out limit and also the application can use the Task Queue provided by GAE wisely to process an http request that needs a longer time to process. According to Malawski et al. (2013), the Task Queue allows to create tasks which can be executed asynchronously in the background and the time out limit for each task executed by the Task Queue is 10 minutes. The next section discusses the restrictions on uploading a file to the GAE sandbox.

In this section we will discuss the inability to upload files to the GAE Sandbox and provide alternate ways to overcome this limitation. Hexiao et al. (2012) explain a successful migration of the online course application written in Java from the traditional network to the GAE Cloud. However Hexiao et al. (2012) acknowledge that it is difficult to move the existing application written in Java to the GAE and one of the challenges faced is the inability of the GAE to upload the files over the internet to the GAE file system. Since the GAE doesn't support storing the files they have to cancel some of the functionalities such as home work uploading and grading features. According to Prodan & Sperk (2013), Zahariev (2009) and Buyya, R., Vechhiola, C. & Selvi, S.T (2013), the applications running in the GAE are prevented from storing files to the GAE file system in order to secure the GAE Infra structure. Gu et al. (2011), Yin et al. (2011) and Buyya, R., Vechhiola, C. & Selvi, S.T (2013) state that the GAE runs an application within a sandbox. An application running within a sandbox is heavily restricted from accessing disc and memory of the host machine as well as prevented from making network calls to external machines. Sandboxing is achieved by limiting the application's access to the Operating system calls. Gu et al. (2011) opinions that even though the application running in the sandbox has limited access to the underlying host infrastructure sandboxing is beneficial such as the sandbox allows the GAE to run instances of applications on multiple servers accross the network, start and stop these servers to meet traffic demands etc. In short these limitations are designed to achieve some specific goals.

The restriction of uploading files to the GAE could be overcome by storing the files to the Datastore or to the Blobstore. Google Developers (2013a) suggest the use of predefined APIs called Blobstore APIs for handling the storage of files to the Blobstore. The acronym API stands for Application Programming Interface, which are interfaces available to the application for performing functions. The acronym Blob stands for Binary large Object. When the user uploads a file to the application via a form, the application can use the Blobstore API's to create a blob object in the Blobstore from the contents of the file and store the key returned by the Blobstore API's. The key can be used to access the file from the Blobstore in the future. According to Dewan & Hansdah (2011), Google's Blobstore is capable of storing blob objects of size up to 2 GB(Gigabyte). In addition, to storing the files to a Blobstore, it is also possible to store small sized files into the Datastore. The Blob data type is supported by Datastore and the files of limited size can be stored into the Blobproperty of the Datastore.

To conclude, the limitations associated with GAE, such as request processing time out and in ability to write the GAE file system, can be solved by following the recommendations from Martins (2012) and using Blobstore APIs respectively. Therefore solving the GAE limitations helps to efficiently migrate the Java/JEE application to the GAE. In the next section Google App Engine Datastore Limitations, we will discuss the limitations of the GAE Datastore.

2.5 Google App Engine Datastore Limitations

In this section we will discuss the limitations of GAE Datastore, delve deeper into the concepts of GAE Datastore to have a better understanding of how Datastore works and finally provide some solutions to overcome these limitations. Most of the journal articles compare traditional relational database management systems to the GAE Datastore and consider that the GAE Datastore has limitations

According to Vu & Asal (2012), the features supported by the traditional relational database management systems are not supported by the GAE Datastore. Vu & Asal (2012) point out that triggers, stored procedures, transactions, indexing, sequencing, joins and predefined functions such as MIN, MAX, COUNT Etc are supported in traditional database management systems, but these features are not supported by GAE Datastore. Vu & Asal (2012) also mention that the GAE Datastore does not support transactions, whereas, Bunch et al. (2010) and Yin et al. (2011) state that the GAE Datastore provides support for transactions. Google Developers (2013e) confirm that GAE Datastore supports transactions. According to Google Developers (2013h), it is possible to group entities to belong to an entity group. The entities in such an entity group can be manipulated in a single transaction. In short, GAE Datastore supports ACID transactions like an RDBMS. The term ACID stands for Atomicity, Consistency, Isolation and Durablility, which are transaction attributes that guarantees the reliability of RDBMS transactions. Vu & Asal (2012) and Chauhan & Babar (2012) point out that table joins are not supported. The table join is a SQL clause which is used for fetching data that spreads across multiple tables.Google Developers (2013b) and Kotecha et al. (2011) state that the Datastore does not support table joins. Kotecha et al. (2011) rule out the need for table joins as the join table data is represented as an aggregate of the related tables. Tran et al. (2011) state that NoSQL systems do not support JOIN operations and an application that uses an RDBMS is migrated to a Cloud Platform which provides a NoSQL sytem it is required to modify the application code to compensate for the lack of features such as JOINS in order to achieve same functionalities and operations of the application before migration. However, In my opinion even though the table joins are not supported by GAE Datastore, it is possible to have a workaround solution to achieve the same results as that of a table joins. The workaround solution in this scenario will be forcing each entity to maintain a property that holds a value which is the equal to the key of another another enitity and also modify the application code so that when an entity is loaded the reference entity is also loaded from the Datastore.

In order to understand why many traditional relational database management features are not supported in the GAE Datastore, a brief understanding about the Datastore is required. According to Buyya, R., Vechhiola, C. & Selvi, S.T (2013) and Lei Hu (2012) the Datastore is built over the Google's proprietary data storage system named Bigtable. Sanderson (2009) agrees that Bigtable is the basis of GAE Datastore. Yin et al. (2011) and Ramanathan et al. (2011) state that the GAE uses Bigtable as the database system for storing the application data and it is built on top of the Google File System (GFS),Chubby service and SSTable. Ramanathan et al. (2011) give a more detailed insight into the Bigtable architecture and state that the Bigtable is different from the traditional relational database management systems and it is a scattered, distributed, persistent multidimensional sorted map. Ramanathan et al. (2011) point that, the Bigtable along with GFS has been designed and engineered to maintain a high performance and availability. The Bigtable can have any number of tables and the data in the tables is indexed using a combination of row and column names. Bigtable does not allow direct querying of the values and requires querying the index tables first locating the data from the Bigtable. Sanderson (2009) states that the GAE Datastore always use an indexing mechanism for retrieving data and an index is maintained for each queries the application is going to perform. Google Developers (2013b) state that the GAE Datastore is flexible enough to allow the application developers to create an index table for a combination of properties. According to Sanderson (2009), the GAE can spread large amount of data and indexes accross many machines and get the results back from all these machines without an expensive aggregate operation. Sanderson (2009) mentions that the indexing strategy has major drawbacks and acknowledges that the query engine of the Datastore is weak when compared to those of the relational Database systems. The query engine of the Datastore cannot execute sophisticated queries like that of RDBMS, however it is suitable for executing simple queries and returning fast results. This is the main reason why GAE Datastore do not support features which are offered by the relation database management systems. However application developer can use workaround solutions to achieve RDBMS features which are not supported by the Datastore. The Google Developers (2014k) confirm the limitations of the Datastore and mentions that Datastore query engine do not provide support for joins and agregate queries which are normally supported by other database technologies. The Google Developers (2014k) also provide a detailed list of features which are not supported.

Gregorio, J. (2008) recommends the use of Sharding counters to achieve the relational database management feature COUNT in the GAE Datastore. The COUNT function is used to get the number of records in a table or the count of non null values in a column. The Datastore do not support functions such as MIN and MAX. The MIN function retrieves the smallest value in the specified column and MAX function retrieves the biggest value in the specified column. The workaround solution could be executing a query on an entity kind with a sort order as ASC or DESC on the entity property returning just a single entity. Da-sheng & Sheng-yu (2010) and Li et al. (2010) explain about the Database stored procedures and Triggers. According to them, a stored procedure is comprised of a set of SQL statements as well as optional flow control statements and resides in the database. They also explain that a Trigger is a special kind of stored procedure that is automatically executed when the data in the table is modified due to an insertion, deletion and updation of records in the table. They state that stored procedure is compiled during the creation and it can execute faster when compared to SQL queries when called from an application. However the GAE Datastore do not support stored procedures and triggers. In my opinion the work around solution will be developing classes that contains methods that can access the Datastore and calling these methods whenever it is appropriate. Another feature which is not supported by the Datastore is sequencing. Some Database management systems support creating Database sequence objects which are used to generate unique identifier values. These values normally may be used as primary key values in tables. Each time a value is retrieved from the sequence object it increments the value stored internally by a predefined count so that each time a new value is generated on accessing the sequence object. Since the GAE Datastore do not support the creation of sequence object, the workaround solution to support database style sequencing is to create a new datastore entity that belongs a new Entity Kind and create a utility function. The newly created entity can have one or more properties which are initialized to numeric values. The property value will serve as the unique sequence value. Whenever the application needs a unique sequence value, the utility function can be invoked which will read the entity from the Datastore and returns the value of the entity property. The utility function also increment the property value by a count and update the entity in the Datastore so that on accessing the entity property each time returns a new value.

To conclude, the Datastore is different from the traditional RDBMS and the Datastore has limitations in supporting some features of RDBMS. Some of these limitations could be overcome by providing workaround solutions as well as following suggestions from Google Developers (2013b) and Gregorio, J. (2008) thereby enabling the Java/JEE application to efficiently migrate to the GAE.

2.6 Summary

The background for the research has considered the challenges such as Programming language incompatibility issues, Database Migration Issues, Google App Engine limitations and Google App Engine Datastore limitations that may be encountered during the migration of Java/JEE application to the GAE. In short, Programming language incompatibility issues can be solved by using only white listed Java classes as well as replacing any unsupported class with alternative Java classes. Database Migration Issues can be solved by the use of bulk loader tool or develop a Database Migration Tool that can migrate the application data from the traditional infrastructure to the GAE Datastore in the PaaS Cloud. The request processing time out limitation of GAE can be efficiently solved by following the recommendations from Martins (2012). The inability to upload files to GAE file system can be efficiently solved by either using Blobstore APIs or storing files to the Datastore instead of storing files to the GAE file system. Google App Engine Datastore limitations can be efficiently solved by using the indexes, reference properties for supporting join operations, Sharding counters wisely.

The background for the research has found that some of the limitations that are pointed out by journal articles are in fact designed or characteristics of the GAE platform. However, the Google Developers acknowledge that there are limitations with the GAE and the Datastore and recommend solutions to overcome most of these limitations. However, further research is highly recommended for the following reasons.

The traditional RDBMS have been around since 1980's, it's powerful, continues its prominence and is an established technology. Migrating the application to the GAE may not be easy as it involves replacing the established RDBMS system with Datastore, which is relatively new and lacks support for some RDBMS features. In addition fitting the Java/JEE application to the GAE infrastructure may also pose major challenges as GAE supports only a subset of Java classes in the Java Standard Library. The research problem is highly important because most journal articles state that the migration of existing applications are difficult and challenging and there are many Java/JEE applications running on organization infrastructures. Moving these applications efficiently to the GAE would help the organizations to save costs and leverage the GAE infra structure to achieve improved application performance.

The research thesis is of the opinion that a Java/JEE prototype application can be efficiently migrated to a Google App Engine PaaS Cloud but this requires a deeper understanding about the working of Google App Engine as well as a different perspective of thinking from traditional computing to Cloud Computing. Moreover the migration of the application and data should be handled in a organized manner. Currently there is no well defined migration process for moving existing applications to GAE. The research thesis has also identified the need to develop efficient tools to resolve some open issues such as identifying the GAE unsupported JRE classes present in the application, identifying the SQL queries in the Data access layer of the application and migrating the application data from the RDBMS of the application to the GAE Datastore. In this thesis paper, we present an efficient and standard software methodology in coordination with the tools developed to efficiently migrate an existing Java/JEE applications to GAE Platform.

In the next chapter Design we describe the proposed software methodology as well as the design and functional requirements for the tools which are developed to support the methodology.

Chapter 3

Design

This chapter discuss the proposed software methodology as well as the design and functional requirements of the Java Source Code Analyzer Tool and Database Migration Tool. These tools are developed in order to support the software methodology. The proposed software methodology is an organized way to efficiently migrate the Java/JEE application to the GAE Platform. The software methodology is composed of a series of steps.

1. The first step is to identify the usage of GAE unsupported JRE classes in the Java sources files of the Java/JEE application. This is achieved with the help of a Java Source Code Analyzer Tool that can scan the application's Java source files for the presence of GAE unsupported JRE classes. The scan results generated should contain information such as the names of the Java classes that contains the GAE unsupported JRE classes, the various ways in which an GAE unsupported JRE classes are used (whether used as interface, superclass or declared as field/method variable), the exact locations in the Java source files where GAE unsupported JRE class are present and refactoring guidelines/hints to follow to remove the identified GAE unsupported JRE classes. The Java Source Code Analyzer Tool is designed and developed to perform this step.

2. The second step uses the scan results generated from the step 1. In this step the detected GAE unsupported JRE classes in the application are replaced with solutions that are in conformance with GAE platform. A good example is an application Java class that has a functionality which requires storing a file to the file system using Java FileWriter API's. The GAE do not permit an application to use Java FileWriter API class for creating files on the GAE infrastructure. So in this scenario the solution is to modify the code to replace the FileWriter API's

with Blobstore API or Datastore API's for storing the files to the Blobstore or GAE Datastore.

3. The third step involves migrating the data from the existing RDBMS that is used by the application to the GAE local Datastore on the development environment for testing. A Database Migration Tool should be designed to perform data migration activities.

4. The fourth step involves modifying the data access components of the application to support the GAE datastore. This step involves replacing the SQL queries present in the Java classes of data access layer with the Datastore APIs. The Java Source Code Analyzer Program designed earlier has an additional functionality to scan for the usage of SQL queries in the Java source files of the Data access layer and generate scan results. The scan result should contains information such as the names of Java classes, the names of the methods in which SQL queries are present, the line numbers in the Java files at which the SQL query strings are declared and also recommends the equivalent Datastore operations to replace the SQL queries.

5. The fifth step is about testing the application in the local development environment and fixing any bugs or issues that are found during the testing phase.

6. The sixth step involves moving the re-engineered application from the local development environment to the GAE platform using the tools provided by GAE SDK.

7. The seventh step involves using the Database Migration Tool once more to extract data from the Database to the remote Datastore in the GAE Cloud.

   The Figure 3.1 shows a pictorial representation of these steps which are employed in the software methodology.

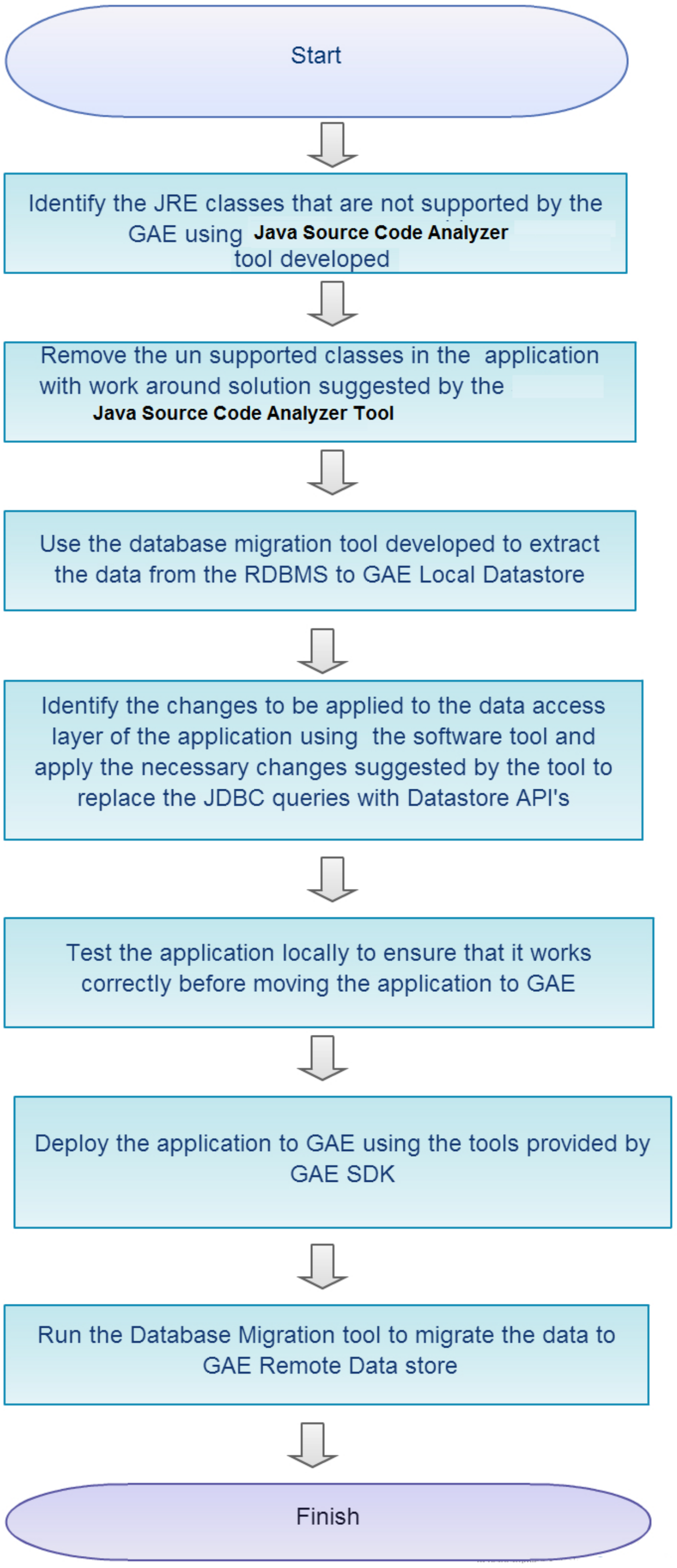
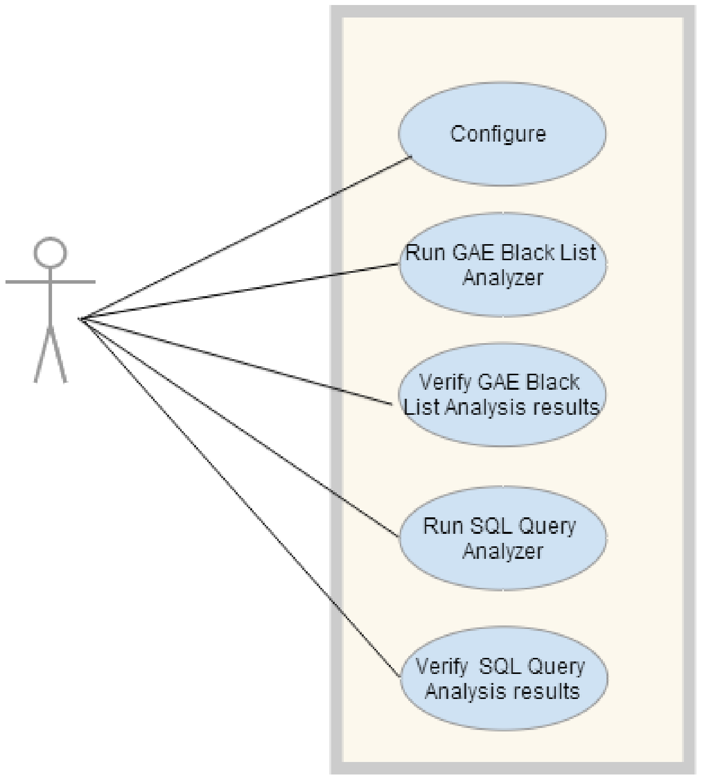

Start
Identify the JRE classes that are not supported by the GAE using Java Source Code Analyzer tool developed
Remove the un supported classes in the application with work around solution suggested by the Java Source Code Analyzer Tool
Use the database migration tool developed to extract the data from the RDBMS to GAE Local Datastore
Identify the changes to be applied to the data access layer of the application using the software tool and apply the necessary changes suggested by the tool to replace the JDBC queries with Datastore API's
Test the application locally to ensure that it works correctly before moving the application to GAE
Deploy the application to GAE using the tools provided by GAE SDK
Run the Database Migration tool to migrate the data to GAE Remote Data store
Finish

Figure 3.1: The steps of the software methodology for migrating Java/JEE applications to GAE Cloud.

Next we will present the main issues which impede the migration of Java/JEE applications to GAE.

1. Currently there are no mature tools available to identify the Java classes in the application that contain both GAE unsupported JRE classes and SQL queries.
2. The Absence of an automated Database Migration Tool that can migrate the application data from the RDBMS to the GAE (local or remote)Datastore.

In order to efficiently solve the first issue we propose to develop a Java Source Code Analyzer Tool. The tool should provide interfaces to identify the usage of JRE classes which are not supported by the GAE Platform and SQL queries in the applications Java files. The Figure 3.2 describes the user's interaction with the Java Source Code Analyzer Tool.



Configure
Run GAE Black List Analyzer
Verify GAE Black List Analysis results
Run SQL Query Analyzer
Verify SQL Query Analysis results

Figure 3.2: UML USE Case Diagram describing user interaction with the Java Source Code Analyzer Tool

On invoking the interface to find occurence of JRE classes which are not supported by GAE platform, the Java Source Code Analyzer Tool scans the application's Java source files to detect the usage of GAE unsupported JRE classes. On successful completion the results are presented to the user. Using the scan results, the user can easily re-engineer the application code to remove the GAE unsupported JRE classes with recommended work around solutions.

On Invoking the interface to find the presence of the SQL queries, the tool scans through the data access layer Java files in the project and identifies the usage of SQL queries that performs CRUD operations on the Database tables. The user can refer to the scan results and easily replace the SQL queries with Datastore API's. The scan results also should display a rule table which the developers can refer to get an idea how the JDBC APIs and Datastore APIs are mapped.

The Figure 3.3 shows an overview of the components of Java Source Code Analyzer Tool, the input parameters, the output results and the process flow. The main modules or components of the Java Source Code Analyzer Tool are the GAE BlackList Analyzer Program, SQL Query Analyzer Program and rule repository. The inputs to both the GAE BlackList Analyzer Program and SQL Query Analyzer Program are the Java Source files of the application. The rule repository contains the rules that aid to refactor the application and it is used by both GAE BlackList Analyzer Program and SQL Query Analyzer Program.

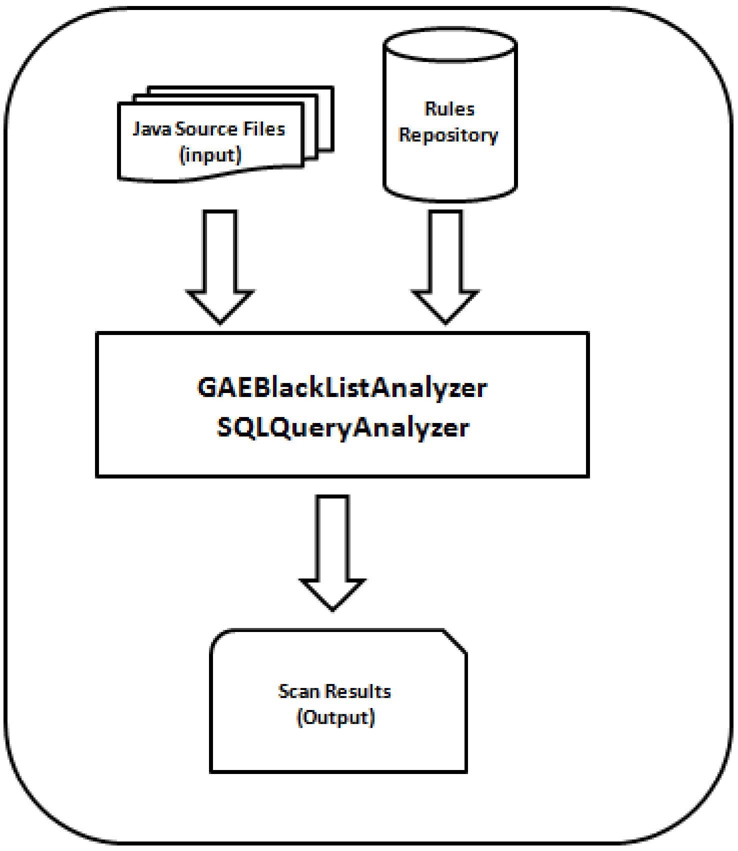


Figure 3.3: The main components of Java Source Code Analyzer Tool

The output of GAE BlackList Analyzer Program is the scan results containing the list of Java classes in the application that uses GAE unsupported JRE classes to perform their function. The scan results also provide additional informations such as line number at which an GAE unsupported JRE class appears, whether the GAE unsupported JRE class is used as a superclass, or as an interface or as a field variables within a class or as a local variable in a method etc. Finally the scan results also contain the work around solutions in order to replace the GAE unsupported JRE class from the application so that there will be no change in the application's functionality after migration to GAE Platform. The work around solutions in the scan results are fetched from the rule repository.

The output of SQL Query Analyzer Program is the scan results containing the list of Java classes that have SQL query strings declared in the methods. The scan results provides information about the line number at which a query string is declared in the method, the kind of operation the SQL query performs (whether it corresponds to create, read, update or delete Database operation). The scan result also provides some generic information from the rules repository or displays a rule table that shows how to replace a SQL query with equivalent Datastore API's. The output of both GAE BlackList Analyzer Program and SQL Query Analyzer Program are intended to the developers to aid the code refactoring.

Having discussed about the Java Source Code Analyzer Tool components we discuss about the Database Migration Tool. The second issue related to data migration can be solved by developing an efficient automated Database Migration Tool. The Database Migration Tool that is to be developed should be able to automate the extraction of data from the RDBMS used by the application to GAE application local Datastore or to GAE application remote Datastore. The research has identified that the Database Migration Tool is the most crucial to achieve an efficient migration of the application to the GAE.

The Database Migration Tool that will be developed should run with minimal manual configuration. The tool should be able to extract data from any database used by the application irrespective of the database vendor. It is also important that the tool should use concurrency techniques to optimize the speed of data migration. The Figure 3.4 describes the user's interaction with the Database Migration Tool.

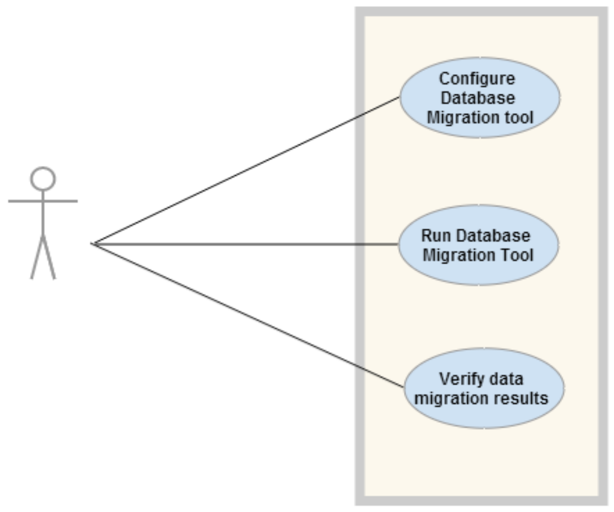

Configure Database Migration tool
Run Database Migration Tool
Verify data migration results

Figure 3.4: UML USE Case Diagram describing the user interaction with the Database Migration Tool

The tool will have two main components or modules namely a Data Exporter and Data Importer. The tool provides an interface called migrate that can be invoked to start the data migration process. The Figure 3.5 shows the overall component diagram.

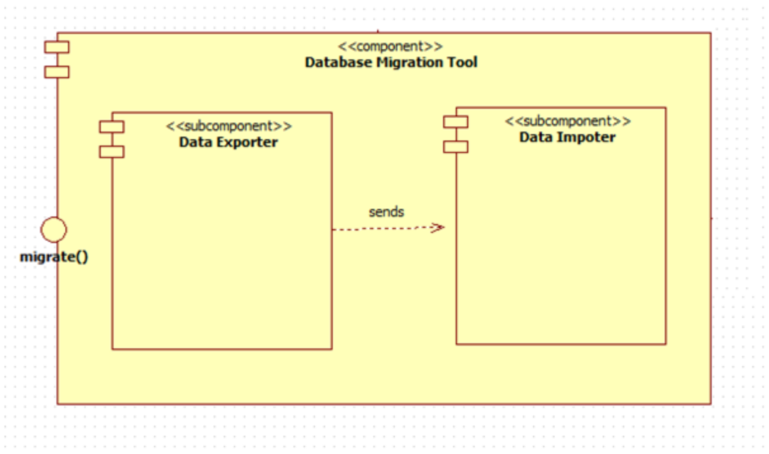


Figure 3.5: UML Component Diagram of the Database Migration Tool

The Database Migration Tool is designed to run a DataExporter and DataImporter thread. Initially the main program thread of the Database Migration Tool creates and starts the DataImporter thread. The DataExporter thread maintains a thread pool object containing a fixed number of worker threads. The number of worker threads in the thread pool is configurable. Each worker thread in the thread pool at a time exports the records in the database table to a set of CSV files. Once a worker thread finishes the task of exporting a table, the worker thread is reused to export another table. Once the worker threads in the thread pool exports all the tables to CSV files the DataExporter thread signals back to the main program thread indicating that all tables are successfully exported. The main program thread then creates and starts a DataImporter thread. The DataImporter thread also maintains a thread pool object containing a fixed number of worker threads. The number of worker threads in the thread pool is configurable. Each worker thread in the thread pool at a time is assigned the task to read the CSV files related to a specific table to create Datastore entities in the GAE local or remote Datastore based on configuration. The worker threads can use remote API's provided by the GAE SDK to access an GAE web application's Datastore. Once the worker threads in the thread pool imports all the data in the CSV files as entities into the Datastore, the DataImporter thread signals back to the main program thread indicating that all data in the CSV files are imported to GAE Datastore. The main program thread resumes execution to completion.

It is also important that the Database Migration Tool logs the results of the execution of these components to a persistent storage. The Database Migration Tool should log data exporter execution details such as the names of the tables that have been exported,the number of rows in each table that have been exported to the CSV files, the status of the exportation(ie whether a table export is successful or not) and the date of execution.

It should also log data importer execution details such as the Kind of entities that are created in the Datastore, the number of entities that has been created for each Entity Kind, the status of the importation and the date of execution. The following diagram shows an overview of the working of Database Migration Tool.

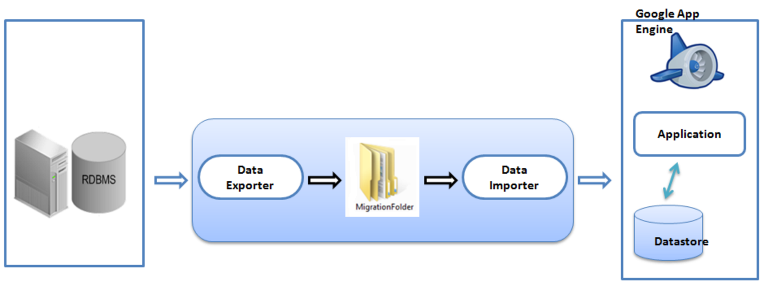


Figure 3.6: Working of the Database Migration Tool

Next, we present "Spring Petclinic" one of the applications used in this thesis to demonstrate and validate our software methodology. The Figure 3.7 shows the deployment diagram for the "Spring Petclinic" application deployed in a traditional way.

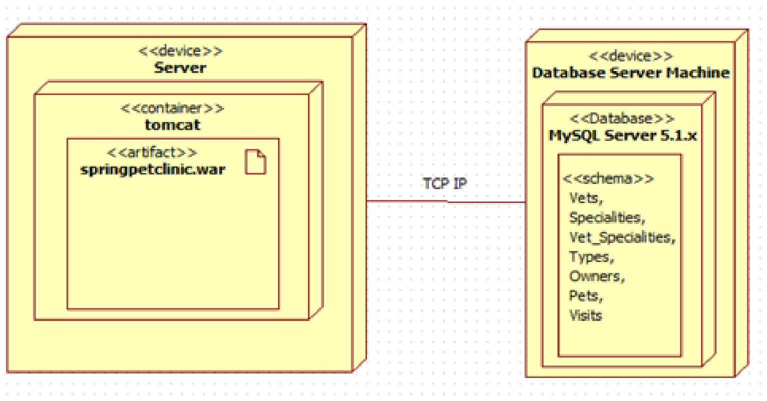


Figure 3.7: Deployment diagram of the 'Spring Petclinic' application in a traditional IT network.

In the Figure 3.7 an application is deployed on a web server that is running on a machine and connects to an RDBMS Database. In the RDBMS the application data will be stored as records in the tables. The application on successful migration to the GAE will have a different architecture. The Figure 3.8 shows the new deployment architecture of the "Spring Petclinic" application after a successful migration.

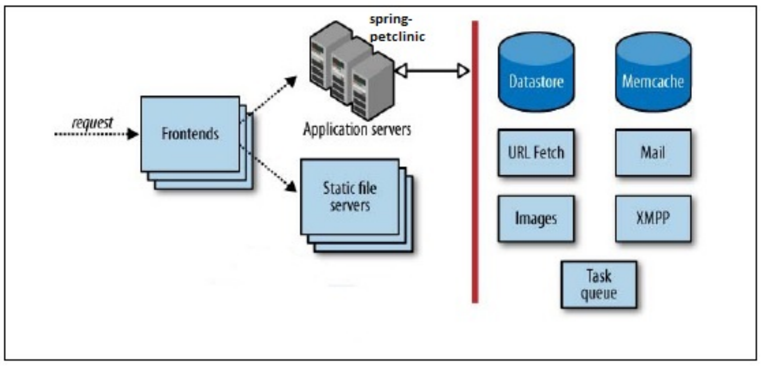


Figure 3.8: Deployment diagram of the ‘Spring Petclinic’ application on Google App Engine.

After migration the “Spring Petclinic” application will be running over the GAE infrastructure and have the following architecture as shown in the Figure 3.8. The “Spring Petclinic” application instances will be running on the application server pool and will be using GAE Datastore for storing application data instead of using RDBMS. However in the GAE Datastore, the “Spring Petclinic” application data is stored as entities or objects. Each entity belongs to a particular Kind. The entities of a Kind are analogous to records of table in the RDBMS. The “Spring Petclinic” application is also free to use other platform specific features provided by the GAE such as Memcache, Mail, URL fetch, Task Queues etc.

3.1 Summary

In this chapter we described the steps involved in the proposed software methodology which is designed for migrating Java/JEE applications to the GAE Platform. This chapter also discusses about the functional requirements, design, architecture and working of both the Java Source Code Analyzer Tool and Database Migration Tool.

In the next chapter we will be discussing more in detail about the underlying technical implementation of the Java Source Code Analyzer Tool and Database Migration Tool.

Chapter 4

Implementation

This Chapter describes the implementation details of the Java Source Code Analyzer Tool and Database Migration Tool that are developed to support the software methodology for migration. Both tools are developed using the API's provided by Java Standard Edition 6. First we will discuss about the Java Source Code Analyzer Tool before we proceed to the Database Migration Tool.

4.1 Java Source Code Analyzer Tool

The Java Source Code Analyzer Tool is comprised of two main programs, namely the Google App Engine BlackList Analyzer Program and SQL Query Analyzer Program. Both the Google App Engine BlackList Analyzer Program and SQL Query Analyzer Program are implemented using Java Compiler API's. In this section we discuss about the usage of Java Compiler API's classes which are profoundly used for developing the Google App Engine BlackList Analyzer Program and SQL Query Analyzer Program.

The Google App Engine BlackList Analyzer Program maintains a repository of JRE classes which are unsupported by GAE and detects if these unsupported JRE classes are used within the application. These unsupported JRE classes may be used in a Java application in different ways and must be removed from the application before it is deployed to GAE Platform. These GAE unsupported classes may be used in the application's Java source files such as declared as a field variables within the Java classes, used as as a super class, used as a method parameters, declared as return type of method, declared as a local variable within a method etc. The SQL Query Analyzer Program checks the use of SQL queries within the methods of all Java classes which are present in application's the Data Access Layer. Both the programs, helps the developers to refactor the application efficiently so that an existing application can be run on GAE Platform. The developers can run both programs and identify which part of the application code to be refactored.

The next paragraph gives a brief description how the Compiler API's are used for the Java Source Code Analyzer Tool as well as the high level architecture of both the Google App Engine BlackList Analyzer Program and SQL Query Analyzer Program of the Java Source Code Analyzer Tool.

The Compiler API's allows to access the Java Compiler from a Java Program (Oracle (2011)). The API's to access a Java compiler is present in the `tools.jar` file which is a part of JDK. The `tools.jar` file contains classes which can be used by Tools and Utility Programs. The Java compiler works closely with the `StandardJavaFileManager` class. The `StandardJavaFileManager` Class has a method `getJavaFileObjects` which takes input as the list of File objects representing Java source files to return a list containing objects of class `JavaFileObject`. A `JavaFileObject` is an abstract representation of Java source file which can be on the disc, memory, database etc. The next step is to obtain a `CompilationTask` object from the Java compiler object by passing the list containing objects of `JavaFileObject` class. The `CompilationTask` object represents a future compilation task and can compile a set of source files by invoking the method `call()` on the `CompilationTask` object. Before invoking the `call()` method on the `CompilationTask` object an `annotation processor` object is registered. On invoking the `call()` method on the `CompilationTask` object the Java compiler calls the `process` method of the annotation processor object which was registered earlier before it begin to compile the Java source files. The `process` method is supplied with an object of class `RoundEnvironment`. The `RoundEnvironment` object can be used to get a set of `Element` objects representing the instances of `JavaFileObject` that was passed to the `CompilationTask` object. The `Element` is a Java type that can be used to represent a class, package, interface or variable. Each `Element` object is accessed from the Element set and it's representation as a tree of nodes is obtained using Java Tree Compiler API's. For example an element representing a Java class can be represented as a tree of nodes. Each children in the nodes represents a meaningful construct. In otherwords a Java class can be represented as a Class tree containing method declarations are represented as MethodTrees, variable declarations are VariableTrees etc. The tree representation of the element is then passed to a custom tree visitor object. The custom tree visitor object's class extends a built-in Java Tree visitor class named `TreePathScanner`. The `TreePathScanner` class visits all the child nodes in a tree. The custom tree visitor class override the appropriate methods from the built-in `TreePathScanner` class in order to process a language construct such as a class, method, variable declaration etc. The overridden methods will contain the processing logic such as whether an application Java class has declared the GAE unsupported classes as a field variables or used as super classes or used as method parameters,or as method return types, or used as method local variables, whether a String variable declared within a method of Java class contains an SQL query string etc. The overridden methods may also use `Java Reflection API's`, `URLClassLoader` class and other utility classes for its functioning. The Reflection API's are used to examine or modify the run time behaviour of applications in the virtual machine and the URLClassLoader is used for loading classes.

Having described about the usage of Compiler API's for implementing the programs of Java Source Code Analyzer Tool, next we will explain it clear with the help of the program code that is developed for Google App Engine BlackList Analyzer Program. The program code 4.1 shows obtaining a Java compiler and registering an annotation processor object. The annotation processor object that is used is an instance of class BlackListCodeAnalyzerProcessor which is an application class belonging to Google App Engine BlackList Analyzer Program.

```
// Gets the Java programming language compiler
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
// Get a new instance of the standard file manager implementation
StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null, ↩
    null);
List<File> files = new ArrayList<File>();
files.addAll(FileSearchMain.getProjectFiles(ConfigurationPropertiesFileReader. ↩
    getProperty("project.rootfolder.location")));
if (files.size() > 0) {
// Get the list of containing objects of JavaFileObject class
Iterable<? extends JavaFileObject> compilationUnits = fileManager. ↩
    getJavaFileObjectsFromFiles(files);
// Create the compilation task
CompilationTask task = compiler.getTask(null, fileManager, null,null, null, ↩
    compilationUnits);
// Get the list of annotation processors
LinkedList<AbstractProcessor> processors = new LinkedList<AbstractProcessor>();
processors.add(new BlackListCodeAnalyzerProcessor());
task.setProcessors(processors);
// Perform the compilation task.
boolean result = task.call();
```

Listing 4.1: Accessing a Java compiler object and registering an annotation processor object.

The program code 4.2 shows the process method of the annotation processor object that we registered earlier. Inside the process method we create a visitor object to scan the Element objects inside the Element set. The Element set is obtained from the

RoundEnvironment object and each object in the Element set may represent a package or a Java class. The visitor class contains the overridden methods from the super class TreeScanner and calls these overridden method during the scanning of an Element.

```java
@Override
public boolean process(Set<? extends TypeElement> annotations,RoundEnvironment ↩
    roundEnv) {
  BlackListCodeAnalyzerVisitor visitor = new BlackListCodeAnalyzerVisitor( ↩
      processingEnv);
  Set<? extends Element> elements = roundEnv.getRootElements();
  for (Element element : elements) {
    TreePath treePath = tree.getPath(element);
    if(treePath!=null){
    visitor.scan(treePath, tree);
    }
  }
  return false;
}
```

Listing 4.2: The process method of the BlackListCodeAnalyzerProcessor Java class.

The program code 4.3 shows an overridden method of the visitor class that contains the logic to check for GAE unsupported JRE class. If a blacklisted class is found it will use an object of ProcessingEnvironment provided by the annotation framework to log the errors to the console.

```java
@Override
public Object visitClass(ClassTree node, Object arg1) {
Trees trees = (Trees) arg1;
Element element = trees.getElement(getCurrentPath());
URLClassLoader urlClassLoader = AppEngineAnalyzerUtils.getUrlClassLoader( ↩
    ConfigurationPropertiesFileReader.getProperty("project.rootfolder.location"));
if (AppEngineAnalyzerUtils.isClass(element)) {

  TypeElement clazz = (TypeElement) element;
  // check for JRE super class which is blacklisted
  checkIfSuperClassBlackListed(clazz, urlClassLoader);
  // check if it implements restricted interfaces
  checkIfInterfaceBlackListed(clazz, urlClassLoader);
}
  return super.visitClass(node, arg1);
}
```

Listing 4.3: The overridden method of BlackListCodeAnalyzerVisitor class containing the logic for checking if a class contains GAE unsupported JRE class

The above code samples demonstrates the usage of Java Compiler API's and uses the code excerpts from the Google App Engine BlackList Analyzer Program to explain the concepts. The SQL Query Analyzer Program also has the same architecture as that of Google App Engine BlackList Analyzer Program. The main difference will be implementation of visitor class. The Google App Engine BlackList Analyzer Program has visitor implementation class that overrides the methods from the built-in Java `TreePathScanner` to check if any of the application's Java classes uses GAE unsupported JRE classes where as the visitor implementation class of SQL Query Analyzer Program visits the methods of the data access layer Java classes to check for the presence of SQL Query strings.

The Figure 4.1 shows the UML class diagram representing the dependencies among the important classes in the Google App Engine BlackList Analyzer Program. The most important classes of the Google App Engine BlackList Analyzer Program are GAEBlackListAnalyzerMain, BlackListCodeAnalyzerProcessor and BlackListCodeAnalyzerVisitor. The method `main(String[] args)` of GAEBlackListAnalyzerMain class serves as an entry point for the Google App Engine BlackList Analyzer Program. The execution of the method `main(String[] args)` starts the Google App Engine BlackList Analyzer Program and obtain a compiler object programmatically as well as register an annotation processor object with the compiler object. The annotation processor object registered is an instance of the class BlackListCodeAnalyzerProcessor. The compiler object calls the `process` method implementation of the BlackListCodeAnalyzerProcessor class. In the `process` method an object of the class BlackListCodeAnalyzerVisitor is used to visit the Abstract Syntax Tree notation of the application's Java classes. The methods of the BlackListCodeAnalyzerVisitor class contain the logic to check for the presence of GAE unsupported JRE classes in the application's Java class files.

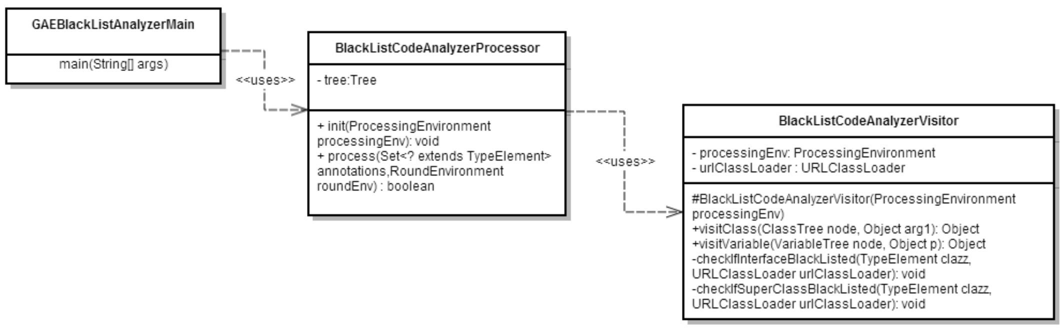

GAEBlackListAnalyzerMain
main(String[] args)
<<uses>>
BlackListCodeAnalyzerProcessor
- tree:Tree
+ init(ProcessingEnvironment processingEnv): void
+ process(Set<? extends TypeElement> annotations,RoundEnvironment roundEnv) : boolean
<<uses>>
BlackListCodeAnalyzerVisitor
- processingEnv: ProcessingEnvironment
- urlClassLoader : URLClassLoader
#BlackListCodeAnalyzerVisitor(ProcessingEnvironment processingEnv)
+visitClass(ClassTree node, Object arg1): Object
+visitVariable(VariableTree node, Object p): Object
-checkIfInterfaceBlackListed(TypeElement clazz, URLClassLoader urlClassLoader): void
-checkIfSuperClassBlackListed(TypeElement clazz, URLClassLoader urlClassLoader): void

Figure 4.1: The UML Class Diagram for GAE BlackList Analyzer Program

The Figure 4.2 shows the UML Class Diagram representing the dependencies among the important classes in the SQL Query Analyzer Program. The main important classes of the SQL Query Analyzer Program are SQLQueryAnalyzerMain, JdbcDAOCodeAnalyzerProcessor and JdbcCodeAnalyzerVisitor. The method `main(String[] args)` of SQLQueryAnalyzerMain class serves as an entry point for the SQL Query Analyzer Program. The execution of the method `main(String[] args)` starts the SQL Query Analyzer Program and obtain a compiler object programmatically as well as register an annotation processor object with the compiler object. The annotation processor object registered is an instance of the class JdbcDAOCodeAnalyzerProcessor. The compiler objects calls the `process` method implementation of the JdbcDAOCodeAnalyzerProcessor class. In the `process` method an object of the class JdbcCodeAnalyzerVisitor is used to visit the Abstract Syntax Tree notation of the application's Java classes. The methods of the JdbcCodeAnalyzerVisitor class contain the logic to check for the presence of SQL Query strings in the application's Java class files.

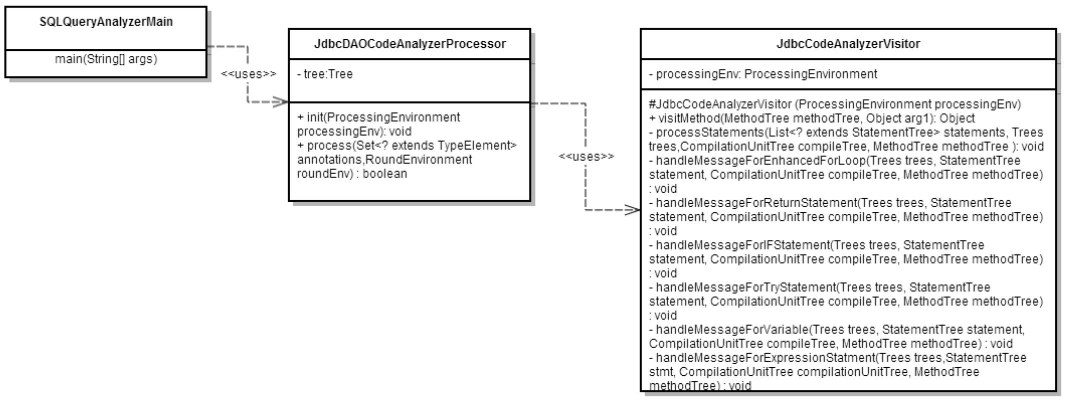

SQLQueryAnalyzerMain
main(String[] args)
<<uses>>
JdbcDAOCodeAnalyzerProcessor
- tree:Tree
+ init(ProcessingEnvironment processingEnv): void
+ process(Set<? extends TypeElement> annotations,RoundEnvironment roundEnv) : boolean
<<uses>>
JdbcCodeAnalyzerVisitor
- processingEnv: ProcessingEnvironment
#JdbcCodeAnalyzerVisitor (ProcessingEnvironment processingEnv)
+ visitMethod(MethodTree methodTree, Object arg1): Object
- processStatements(List<? extends StatementTree> statements, Trees trees,CompilationUnitTree compileTree, MethodTree methodTree): void
- handleMessageForEnhancedForLoop(Trees trees, StatementTree statement, CompilationUnitTree compileTree, MethodTree methodTree) : void
- handleMessageForReturnStatement(Trees trees, StatementTree statement, CompilationUnitTree compileTree, MethodTree methodTree) : void
- handleMessageForIFStatement(Trees trees, StatementTree statement, CompilationUnitTree compileTree, MethodTree methodTree) : void
- handleMessageForTryStatement(Trees trees, StatementTree statement, CompilationUnitTree compileTree, MethodTree methodTree) : void
- handleMessageForVariable(Trees trees, StatementTree statement, CompilationUnitTree compileTree, MethodTree methodTree) : void
- handleMessageForExpressionStatment(Trees trees,StatementTree stmt, CompilationUnitTree compilationUnitTree, MethodTree methodTree) : void

Figure 4.2: The UML Class Diagram for SQL Query Analyzer Program

The Java Source Code Analyzer Tool is available for download from GitHub[1].

Having discussed about the implementation details about the Java Source Code Analyzer Tool, next we will discuss about the implementation details about the Database Migration Tool that is developed.

4.2 Database Migration Tool

The Database Migration Tool is developed using Java concurrency API's to have Multithreading features and also uses the infrastructural support provided by the Spring Framework. Before we proceed it is important to have a basic understanding about the Java Multithreading model. In Java programming language it is possible to create a

[1]https://github.com/prasanthmp500/JavaSourceCodeAnalyzerTool

parallel execution unit by creating a class that implements the Java `Runnable` interface. The class that implements the Runnable interface must implement the `run()` method which represents a concurrent unit of execution. The object of the class that implements Runnable interface is then passed to a built-in Java thread object during creation of the thread object via a constructor method. A thread object is basically an instance of the Java built-in `Thread` class. Invoking the `start()` method on the thread object creates a new thread of execution and calls `run()` method on the Runnable instance that was passed to the thread. Another approach instead of creating a thread objects is to use a factory class named `Executors` to create a thread pool object containing worker threads and submitting the objects of the Java class that implements Runnable interface to the thread pool.

An important mechanism that is used in the Database Migration Tool is the use of Java `CountDownLatch` API's. The `CountDownLatch` API's are the Java built-in synchronization mechanism that allows one or more threads to wait until another set of threads completes their operation. Java provides a built-in class called `CountDownLatch`. A CountDownLatch object can be initialized with a count during the instantiation. Calling the `await()` method on the `CountDownLatch` object causes the current threads to wait until the latch count is zero. Invoking the method `countDown()` on the `CountDownLatch` object cause to decrement the count of the latch object and the waiting threads may resumes execution when the count of latch object becomes zero. In the next section we will discuss about the architecture of the Database Migration Tool and the functioning of the tool.

4.2.1 Database Migration Tool Architecture

In this section we will discuss about the architecture of the Database Migration Tool and the database migration workflow. The tool has two main modules a Data Exporter Module and a Data Importer Module. The Data Exporter module has classes that can handle tasks such as retrieving the records from the database tables, exporting these table records as CSV files on the file system, logging the export status of each table etc. The Data Importer module has classes that can handle tasks such as reading the CSV files from the file system, creating Datastore entities in the GAE Datastore, logging the import status of the creation of entities for each Entity Kind. The following Figure 4.3 shows an overview of the components of the Database Migration Tool and functioning of the tool.

Initially the user starts the main thread by invoking the function `main(String args[])` of the class `SpringDataMigration`. The main thread creates an object of class

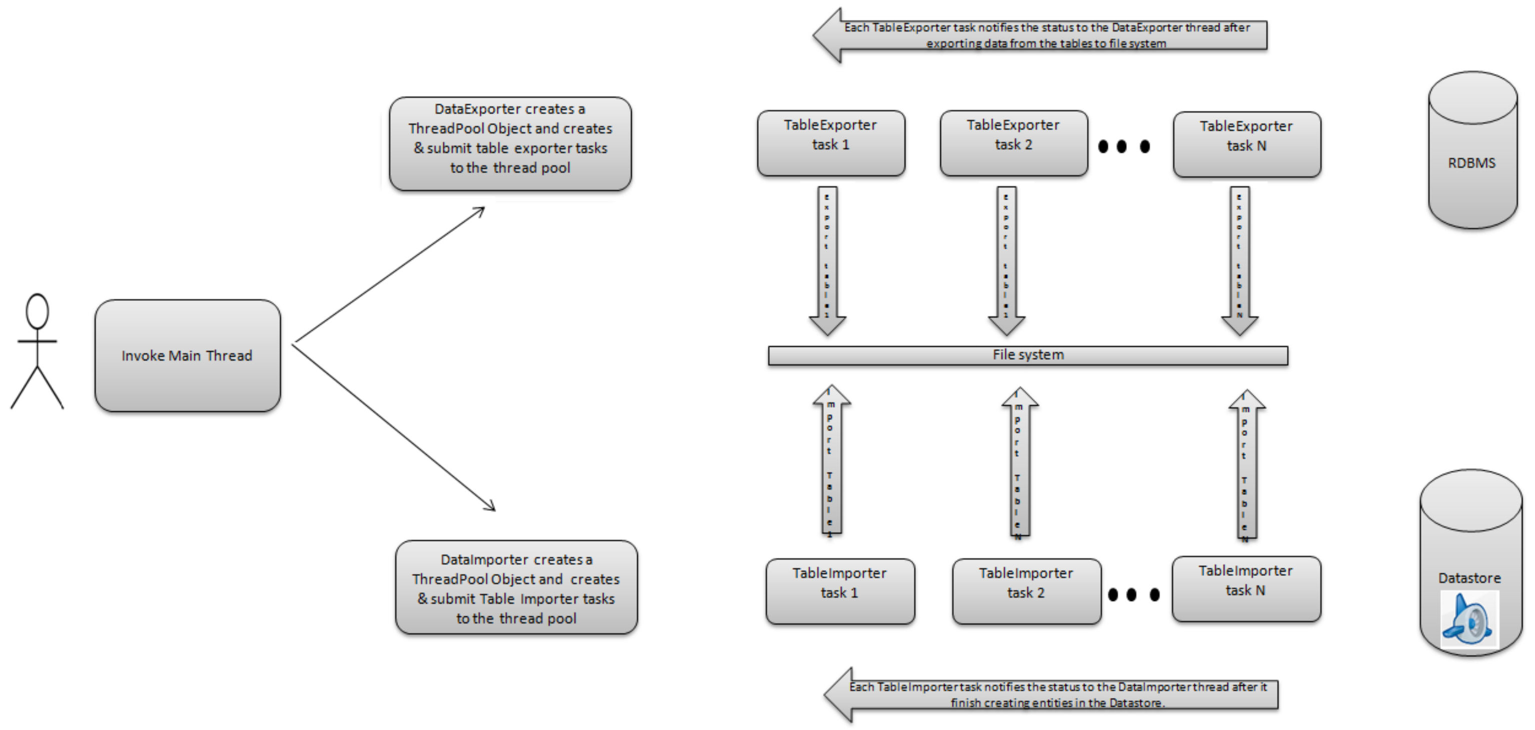

Each TableExporter task notifies the status to the DataExporter thread after exporting data from the tables to file system
RDBMS
DataExporter creates a ThreadPool Object and creates & submit table exporter tasks to the thread pool
TableExporter task 1
TableExporter task 2
TableExporter task N
export table 1
export table 2
export table N
Invoke Main Thread
File system
Import Table 1
Import Table 2
Import Table N
DataImporter creates a ThreadPool Object and creates & submit Table Importer tasks to the thread pool
TableImporter task 1
TableImporter task 2
TableImporter task N
Datastore
Each TableImporter task notifies the status to the DataImporter thread after it finish creating entities in the Datastore.

Figure 4.3: Database Migration Tool main components and workflow

`DataExporter` which implements the Java `Runnable` interface and also creates a Java thread object which is passed with the object of DataExporter class via the constructor method. Invoking the `start()` method on the thread object creates a new thread of execution and calls `run()` method of the `DataExporter` object thereby starting the data export process. The main thread then goes to a waiting state until the exportation of the tables in the Database is completed. It resumes execution once the `run()` method of DataExport object is run to completion indicating that all the tables are exported successfully to the file system. The communication between the main thread and the thread object that executes the `run()` method of the DataExporter object is achieved using a `CountDownLatch` object.

When the main thread is waiting for the thread object to complete, the thread object invokes the `run()` method on the `DataExporter` object. In the `run()` method a thread pool is created using a Java built-in factory class named `Executors`. The number of worker threads in the thread pool is configurable and can be defined in a properties file. The worker threads in the thread pool are assigned the tasks of exporting the tables to CSV files. Each worker thread in the thread pool is assigned the task of the exporting the data from a table at a time. When a worker thread completely exports the table records as CSV files to the file system, the thread is then reused to export other tables. Once the worker threads in the thread pool exports all the tables to the file system, the thread that invoked the `run()` method of the DataExporter object is notified to inform that all table are executed, which in turn notifies the main thread which is in a waiting state. The main thread resumes execution.

The main thread then instantiates an object of class `DataImporter` which implements the Java `Runnable` Interface. The main thread then create a Java `Thread` object and also passes the `DataImporter` object to the thread object via a constructor method during the creation of the thread. Invoking the `start()` method on the thread object creates a new thread of execution and it calls `run()` method of the DataImporter object. The execution of the `run()` method starts the data import process. The main thread then goes to the waiting state once again until it is notified when the creation of Datastore entities from the CSV files is completed. Inside the `run()` method of the `DataImporter` object a thread pool is created and each worker thread in the thread pool is assigned the task of creating the GAE Datastore entities of a specific Kind from a set of CSV files at time. When a worker thread finishes the creation of Datastore entities of a specific Kind, it is reused to create Datastore entities of another Entity Kind from a different set of CSV files. When the worker threads in the thread pool finishes the creation of Datastore entities from the CSV files, they notifies the thread that invoked the `run()` method on the DataImporter object, which in turn notifies the main thread. The main thread that is currently in waiting state resumes execution and terminates.

The program code 4.4 shows the main method of the Java class named `SpringPetClinicDataMigration`, starting the data export process, waiting for the data export process to complete, resuming execution when the data export process is over, then starting the data import process, again waiting for the data import process to finish before resuming the execution to completion of the entire migration process. The main method is basically an entry point of execution of the Database Migration Tool. The program code 4.5 shows the `run()` method of the `DataExporter` class creating a thread pool and submitting the tasks, of reading the records from database tables and creating CSV files, to the thread pool for execution. The program code 4.6 shows the `run()` method of the `DataImporter` class creating a thread pool and submitting the tasks of reading the CSV files and creating GAE enties in the GAE local or remote Datastore to the thread pool for execution.

```
public static void main(String args[]) {
  loadApplicationContext();
  init();
  System.out.println("************************************************");
  System.out.println("Starting Data Migration");

  long time1 = Calendar.getInstance(TimeZone.getTimeZone("GMT")).getTimeInMillis();;

  CountDownLatch latch1 = new CountDownLatch(1);
  DataExporter dataExporterThread = new DataExporter();
  dataExporterThread.setContext(ctx);
  dataExporterThread.setCountDownLatch(latch1);
  Thread t1 = new Thread(dataExporterThread);
  t1.start();
  latch1.await();
  CountDownLatch latch2 = new CountDownLatch(1);

  System.out.println("Exporting Database Tables to CSV files is Completed and now ↩
      Start to Export CSV files to GAE Entities");

  DataImporter dataImporterThread = new DataImporter();
  dataImporterThread.setContext(ctx);
  dataImporterThread.setCountDownLatch(latch2);
  Thread t2 = new Thread(dataImporterThread);
  t2.start();
  latch2.await();
  long time2 = Calendar.getInstance(TimeZone.getTimeZone("GMT")).getTimeInMillis();
  System.out.println("Time in millisecond "+ (time2-time1));
  System.out.println("Completed Data Migration");
}
```

Listing 4.4: The method main of the SpringPetClinicDataMigration class (main thread) coordinating the data export and data import process.

```
public void run() {
  Properties configurationProperties = (Properties) context.getBean("threadPoolPropertiesConfiguration");
  ExecutorService executorService = Executors.newFixedThreadPool( Integer.valueOf( configurationProperties.getProperty("exportThreadPoolSize")));
  List<String> databaseTables = getDatabaseTableNames();
  CountDownLatch exporterCountLatch = new CountDownLatch( databaseTables.size());
  for (String tableName : databaseTables) {
    TableExporter tableExporterBean = (TableExporter) context.getBean("tableExporter");
    tableExporterBean.setTableName(tableName.toUpperCase());
    tableExporterBean.setCountDownLatch(exporterCountLatch);
    executorService.submit(tableExporterBean);
  }
  executorService.shutdown();
  try {
    for (Future<?> future : futures) {
      future.get(); // cause the current thread to wait for the table import tasks to finish.
    }
    exporterCountLatch.await();
    countDownLatch.countDown();
  } catch (InterruptedException e) {
    e.printStackTrace();
  } catch (ExecutionException e) {
    e.printStackTrace();
  }
}
```

Listing 4.5: The run() method of DataExporter.java Class.

```
public void run() {
  Properties configurationProperties = (Properties) context.getBean("threadPoolPropertiesConfiguration");
  ExecutorService executorService = Executors.newFixedThreadPool( Integer.valueOf( configurationProperties.getProperty("importThreadPoolSize")));
  Map<String, String> tableToEntityMap = (Map<String, String>) context.getBean("tableToEntityMapping");
  Set<String> databaseTableNames = tableToEntityMap.keySet();
  CountDownLatch importerCountLatch = new CountDownLatch(databaseTableNames.size() - 1);
  for (String tableName : databaseTableNames) {
    TableImporter tableImporter = context.getBean(TableImporter.class);
    tableImporter.setImporterCountLatch(importerCountLatch);
    tableImporter.setTableToExport(tableName);
    tableImporter.setEntityName(tableToEntityMap.get(tableName));
    tableImporter.setFolderName(tableName.toUpperCase());
    executorService.submit(tableImporter);
  }
  executorService.shutdown();
  try {
    for (Future<?> future:futures) {
      future.get();//cause the current thread to wait for the table export tasks to ↩
          finish.
    }
    importerCountLatch.await();
    countDownLatch.countDown();
  } catch (InterruptedException e) {
    e.printStackTrace();
  } catch (ExecutionException e) {
    e.printStackTrace();
  }
}
```

Listing 4.6: The run() method of DataImporter.java Class.

In the following sections we will explains more about the most important classes of the Data Exporter and Data Importer modules.

4.2.2 Data Exporter Components

The most important Java classes in the Data Exporter module are DataExporter.java and TableExporter.java classes. The Figure 4.4 shows the UML class diagram for the DataExporter and TableExporter classes. Both of these classes implements the Java built-in `Runnable` Interface.

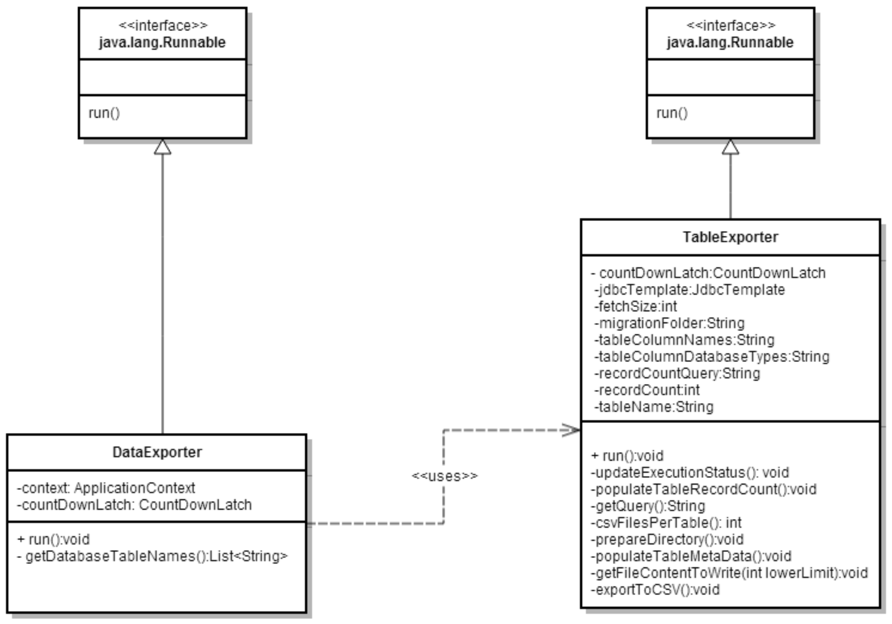


Figure 4.4: UML Class Diagram showing the dependency between DataExporter class and TableExporter class of Data Exporter Module.

The main thread creates an instance of DataExporter class and passes that instance to a newly created thread object via a constructor method. Invoking the `start()` method on the thread object creates a new thread of execution and invoke the `run()` method of the `DataExporter` object that was passed earlier. The main thread waits till the thread object finishes it execution. The main thread goes to a waiting state by calling the `await()` method on an object of `CountDownlatch` class. The same `CountDownlatch` object is also passed to the `DataExporter` object via a property setter method. Before the `run()` method of `DataExport` object finishes to execution, the `countDown()` method is invoked on the `CountDownlatch` object's so that the main thread could resume the exection.

The program code 4.5 shows creation of fixed thread pool within the `run()` method of the `DataExporter` class. Within the `run()` method, the method `getDatabaseTableNames()` is invoked internally to connect to the RDBMS used by the application and returns the list of table names from the database schema. For each corresponding table name in the list, a `TableExporter` instance is created. A `CountDownLatch` object is also created initialized with count equal to the number of tables to be exported. Each `TableExporter` object is set with the table name and also passed a reference of `CountDownLatch` object. Each `TableExporter` object is then submitted to the thread pool by calling the `submit` method as shown in the program code 4.5. The worker threads in the thread pool invokes the `run()` method of the TableExporter objects.

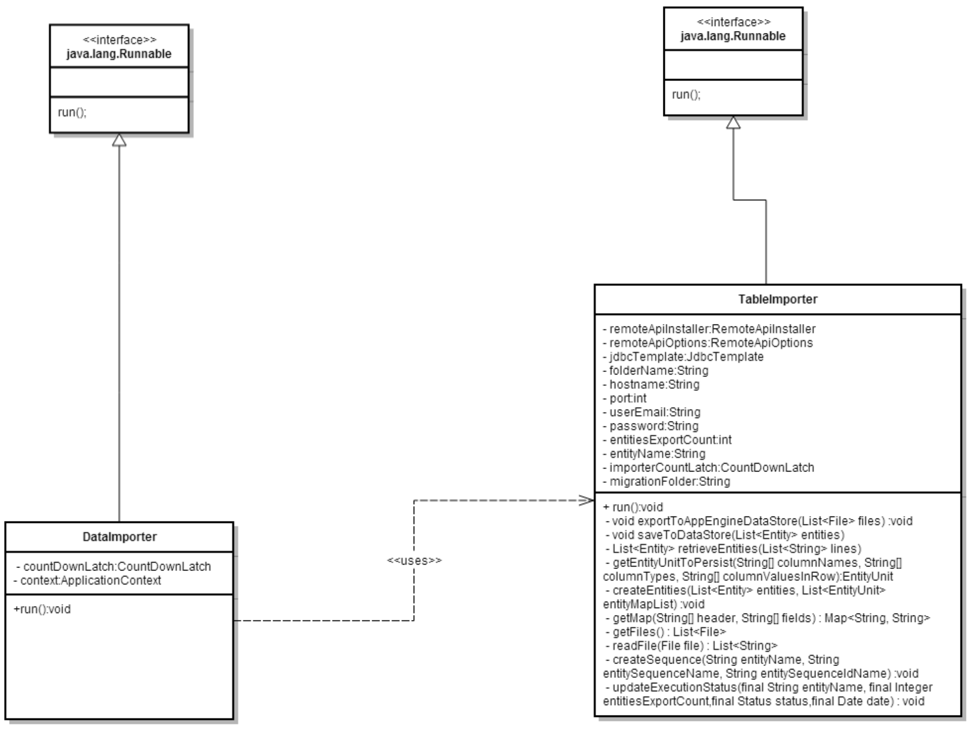
The program code 4.7 shows the `run()` method of the `TableExporter` class containing the logic to export the records in the Table to CSV files in the file system. In the `run()` method it calls main functions such as `populateTableRecordCount()`, `populateTableMetaData()`, `exportToCSV()` and `updateExecutionStatus(String tableName, Integer recordCount,Status status,Date date)`. The method `populateTableRecordCount()` retrieves the count of records present in a table. The method `populateTableMetaData()` gets the lists of column names and column types in the table which will be used later to create Entity properties for the GAE entities. The method `exportToCSV()` fetches the records from table in chunks and creates a CSV file for each chunk of records. The chunk size can be configured in the migration tools properties file named `SpringDatabaseMigration.properties`. The chunk size and the total record count in the table are used to determine the number of CSV files to be created for each table. Once exporting a table is completed the `updateExecutionStatus(String tableName, Integer recordCount,Status status,Date date)` method is called to store the status of the export task by adding an entry in the log table named `data_export_result` such as the number of records exported, status of export and date of export. Finally in the `run()` method of `TableExporter` Object the `countDown()` method is called on the `CountDownLatch` object so that the thread that invoked the `run()` method of `DataExporter` object is notified once the export is complete. When all the tasks for exporting tables are completed, the thread that invoked the `run()` method of the DataExporter object then invoke the `countDown()` method on the `CountDownLatch` object which was passed by main thread to resume the execution of the main thread.

```
public void run() {
  System.out.println("Starting to export Data from Table [ " + getTableName() + " ] ↩
      to CSV files");
  populateTableRecordCount();
  populateTableMetaData();
  try {
    exportToCSV();
    updateExecutionStatus(tableName,recordCount,Status.SUCCESS,new Date());
  } catch (Exception e) {
    updateExecutionStatus(tableName,null,Status.FAILURE,new Date());
  }
  countDownLatch.countDown();
  System.out.println("Finished exporting Data from Table [ " + getTableName() + " ] ↩
      to CSV files");
}
```

Listing 4.7: The run() method of TableExporter class containing the workflow for exporting a table to CSV files

4.2.3 Data Importer Components

The most important Java classes in the Data Importer module are DataImporter.java and TableImporter.java classes. The Figure 4.5 shows the class diagram for the DataImporter and TableImporter classes. Both of these classes implements the Java built-in `Runnable` interface.



<<interface>>
java.lang.Runnable
run();
<<interface>>
java.lang.Runnable
run();
TableImporter
- remoteApiInstaller:RemoteApiInstaller
- remoteApiOptions:RemoteApiOptions
- jdbcTemplate:JdbcTemplate
- folderName:String
- hostname:String
- port:int
- userEmail:String
- password:String
- entitiesExportCount:int
- entityName:String
- importerCountLatch:CountDownLatch
- migrationFolder:String
+ run():void
- void exportToAppEngineDataStore(List<File> files) :void
- void saveToDataStore(List<Entity> entities)
- List<Entity> retrieveEntities(List<String> lines)
- getEntityUnitToPersist(String[] columnNames, String[] columnTypes, String[] columnValuesInRow):EntityUnit
- createEntities(List<Entity> entities, List<EntityUnit> entityMapList) :void
- getMap(String[] header, String[] fields) : Map<String, String>
- getFiles() : List<File>
- readFile(File file) : List<String>
- createSequence(String entityName, String entitySequenceName, String entitySequenceIdName) :void
- updateExecutionStatus(final String entityName, final Integer entitiesExportCount,final Status status,final Date date) : void
<<uses>>
DataImporter
- countDownLatch:CountDownLatch
- context:ApplicationContext
+run():void

Figure 4.5: UML Class Diagram showing the dependency between DataImporter class and TableImporter class of Data Importer Module.

The main thread resumes execution after a successful exportation of the records from the RDBMS to the file system. The main thread creates an instance of `DataImporter` class and passes the instance to a newly created thread object via a constructor method. Invoking the `start()` method of the thread object starts a new thread of execution which invoke the `run()` method of the `DataImporter` object that was passed earlier. The main thread waits till the thread object finishes the execution. The main thread goes to a waiting state by invoking `await()` method on newly created a `CountDownLatch` object that was initialized with a count value of one. The same `CountDownLatch` object was passed to the `DataImporter` object via a property setter method before the `await()` method is called on the `CountDownLatch` object by the main thread. Before the `run()` method of the DataImporter object finishes to execution, the count value of the`CountDownLatch` object's is decremented by calling the `countDown()` method on the `CountDownLatch` object so that the thread object that executes the `run()` method of the `DataImporter` object is notified, eventually notifies the waiting main thread to resume the exection.

The program code 4.6 shows creation of fixed thread pool within the `run()` method of the `DataImporter` class. The `run()` method retrieves a map object containing the key value pairs representing the mapping between the table name in the database and the name of entity kind that will be created in the Datastore. The map object is pre-configured in the migration tools configuration XML to serve as a mapping between the Table in the RDBMS and Entity Kind in the GAE Datastore. The key set of the map contains the names of tables.

For each table names in the key set of the map, a `TableImporter` object is created. A `CountDownLatch` object is also created and initialized with count equal to the number of tables to be imported in to the GAE Datastore. Each `TableImporter` object is set with the table name, the Enity Kind name and a reference to the `CountDownLatch` object. Each `TableImporter` object is then submitted to the thread pool for execution by calling the `submit()` method as shown in the program code 4.6. The threads in the thread pool invokes the `run()` method of the TableImporter objects.

The program code 4.8 shows the `run()` method of the TableImporter class containing logic to read the CSV files related to a specific table and create GAE Datastore entities in the local GAE Datastore or in the remote GAE Datastore based on the configuration provided. In the `run()` method it calls main functions such as `getFiles()`, `exportToAppEngineDataStore(List<File >files)` and `updateExecutionStatus(String entityName,Integer entitiesExportCount,Status status, Date date)`. The method `getFiles()` locates and retrieves the CSV files related to a table from the file system. The CSV files retrieved are then passed to the method `exportToAppEngineDataStore(List<File >files)` which iteratively reads the contents of the CSV files, creates List of objects of appengine Entity class and then uses a `DataStoreService` object to insert the entites into the Datastore by using the GAE provided remote APIs. Once the creation of entities for a specific table is completed the `updateExecutionStatus` method stores the status of the import by adding an entry in the log table named `data_import_result` such as the name of the Entity Kind under which entities are stored, the number of entities imported to GAE Datastore in an Entity Kind, status of import and date of import. Finally in the `run()` method of the `TableImporter` object, the `countDown()` method is called on the `CountDownLatch()` object so that the thread that invoked the `run()` method of `DataImporter` object is notified to indicate that the import process is complete. Before returning from the `run()` method of the DataImporter, the `countDown()` method is called on the `CountDownLatch` object which was passed by main thread in order to resume the execution of the main thread. The main thread then resumes execution, prints a message indicating the data migration from RDBMS to GAE is complete and the database migration process completes. The code developed for the Database Migration Tool can be downloaded from the GitHub [2].

```
public void run() {
  System.out.println("Creating App Engine Datastore Entities for table [ "+ ↩
      tableToExport + " ] from the CSV files");
  List<File> files = getFiles();
  try {
    exportToAppEngineDataStore(files);
    updateExecutionStatus(entityName,entitiesExportCount,Status.SUCCESS,new Date());
    System.out.println("Completed Creation of App Engine Datastore Entities for ↩
        table [ "+ tableToExport + " ] from the CSV files");
    Thread.sleep(1000);
    importerCountLatch.countDown();

  } catch(Exception e){
    e.printStackTrace();
    updateExecutionStatus(entityName,entitiesExportCount,Status.FAILURE,new Date());
  }
}
```

Listing 4.8: The run() method of TableImporter class containing the workflow for importing the data from the CSV files to entities in the Datastore

4.2.4 Data flow diagram

The Figure 4.6 shows the flow of data from the RDBMS to Google App Engine Datastore during the data migration process initiated by the Database Migration Tool. The Database Migration Tool can be configured to export data from the RDBMS to local GAE Datastore provided by GAE development environment or to GAE Datastore in the GAE Cloud.

[2] https://github.com/prasanthmp500/spring-data-migration

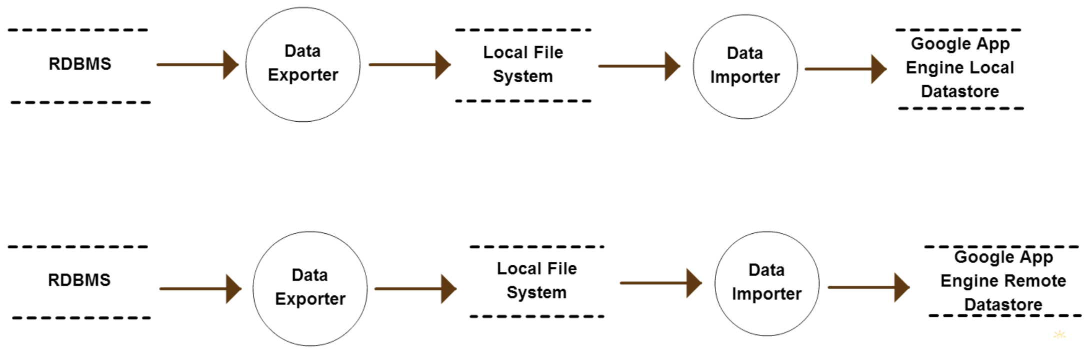


Figure 4.6: Data flow diagram

4.3 Software Life cycle

The development of the tools used for the thesis work followed the waterfall project life cycle model. The waterfall life cycle model has a set of phases that are executed in a sequential order. The waterfall life cycle process begins with a requirement phase, followed by a design phase, implementation phase, testing phase and maintenance phase. The thesis work used the waterfall model due to its simplicity, ease of use and project requirements are known up front. In addition the steps in the software methodology are executed in a sequential manner and cannot be executed in a parallel. The thesis work decided not to use other software life cycle models such as iterative and increment development, prototype, spiral model and agile methodologies as some of these project life cycle models require developing a prototype or iterations of the phases of the waterfall model or are oriented towards reducing risk in the project which are not real concerns with respect to this thesis. The code developed for the both the tools are uploaded to GitHub.

4.4 Main implementation decisions

This section deals with main implementation decisions such as the choice of programming languages, design decisions and frameworks that will be used in the development of the migration tool.

- Java
  The Java Source Code Analyzer Tool and Database Migration Tool will be developed in Java. The main reasons to use Java is that it has support for object oriented programming, multi-threading features and also has wide support for

Compiler API's/ Annotation Processor classes etc that helps to parse through the Java source files.

- Usage of Java Multi Threading and Compiler API's
  The Database Migration Tool is designed to use Multi threading concepts to improves the efficiency of the Database Migration Tool. Multiple threads are spawned simultaneously that perform exportation and importation of the data. The use of threading can speeden the migration process and in addition the threads consumes less resources as threads are light weight processes.

  The Java Source Code Analyzer Tool is developed using both the Java Compiler API's and Compiler Tree API's. The Compiler API's allows the programmer to access to the Java compiler programmatically to compile Java source files. The Compiler Tree API's can be used to obtain an Abstract Syntax Tree (AST) for a Java class and the nodes in the Abstract Syntax Tree can be visited to identify the GAE unsupported features and the usage of strings that represent SQL queries.

- Spring Framework
  The Spring Framework can provide infrastructure support for developing robust Java application easier and faster. The development of the migration tool will be using the infrastructural support provided by the Spring Framework. The Spring Framework provides features such as dependency injection, aspect oriented programming, JDBC support etc. These features can help to reduce programming complexity and efforts as well has handles many low level tasks such as handling database connections.

4.5 Summary

To summarize, in this chapter we have discussed about the implementation details of the tools that are developed to support the software methodology. The main components of Java Source Code Analyzer Tool namely Google App Engine BlackList Analyzer Program and SQL Query Analyzer Program share similar architecture and are implemented with the help of Java Compiler API's, Compiler Tree API's, Java annotation processor classes, Reflection API's etc. The Database Migration Tool has multithreading features and it is architectured using the multithreading API's offered by Java language.

Having described about the implementation aspects of the Java Source Code Analyzer Tool and Database Migration Tool, in the next chapter we will evaluate the software methodology by applying both Java Source Code Analyzer Tool and Database Migration Tool on the sample set of sample Java projects in order to migrate them to the GAE Platform.

Chapter 5

Evaluation

In this chapter we perform an empirical evaluation to validate our software methodology. We design experiments to evaluate each of the two proposed approaches, namely the Java Source Code Analyzer Tool and Database Migration Tool. we conduct the evaluation on two applications namely PetClinic[1] and AjaxCrudJTable[2,3].

The PetClinic is an enterprise level application developed to demonstrate the capability of Spring Framework for building simple and powerful Database oriented applications. The PetClinic application uses MySQL5.x as the Database technology for storing the application data.

The AjaxCrudJTable application is a simple Java web application that is used to demonstrate Database CRUD Operations using jTable. The jTable is a jQuery Plugin which is used to create Ajax based CRUD tables without coding HTML or Javascript. The AjaxCrudJTable application also uses MySQL5.x as the Database technology to store the application data.

the next section we present the evaluation of Java Source Code Analyzer Tool on these sample applications.

[1]https://github.com/spring-projects/spring-petclinic

[2]https://github.com/prasanthmp500/AjaxjTableServlet

[3]http://www.simplecodestuffs.com/ajax-based-crud-operations-in-jsp-and-servlet-using-jtable-jquery-plug-in/

5.1 Evaluation of Java Source Code Analyzer Tool

The Java Source Code Analyzer Tool has two parts namely Google App Engine BlackList Analyzer Program and SQL Query Analyzer Program. First we present the evaluation of Google App Engine BlackList Analyzer Program. The Google App Engine BlackList Analyzer Program is developed to detect the GAE unsupported JRE classes in the application. We compare our approach with Google Plugin for Eclipse. The Google Plugin for Eclipse allows the Java develpers to easily create and deploy applications on App Engine. The Google Plugin for Eclipse also provides real time code validation for ensuring the compatibility of the aplication code with GAE Platform. So we evaluated the Google App Engine Blacklist Analyzer Program versus the popular Google Plugin for Eclipse on the choosen sample applications.

The Table 5.1 shows the total number of GAE unsupported JRE classes identified in the PetClinic and AjaxCrudJTable applications by the Google App Engine Blacklist Analyzer program and Google Plugin.

	# GAE BlackList Analyzer Program	**# Google Plugin**
PetClinic	3	0
AjaxCrudJTable	9	0

Table 5.1: The total number of occurences of GAE unsupported JRE classes identified by GAE BlackList Analyzer Program versus Google Plugin in the sample applications

The Google Plugin failed to identify any GAE unsupported JRE classes even though the PetClinic and AjaxCrudJTable applications contains a few instances GAE unsupported JRE classes in the Java files. The PetClinic and AjaxCrudJTable applications use Java Database Connectivity(JDBC) API's for accessing the Database. However JDBC APIs are not supported by GAE Platform to access the GAE Datastore. The Google App Engine BlackList Analyzer Program provided satisfactory results as it identified the occurences of these JDBC API's in the applications source files.

Next we present evaluation results obtained for the SQL Query Analyzer Program. The SQL Query Analyzer Program identified the usage of SQL queries in the Data access layer of both the PetClinic and AjaxCrudJTable applications. The error logs provided by SQL Query Analyzer Program were used to locate the usage of SQL queries in the Java classes and these SQL queries were easily replaced with Datastore API's. The

Table 5.2 shows the number of SQL queries identified by the SQL Query Analyzer Program in both PetClinic and AjaxCrudJTable applications.

	# SQL queries identified by SQL Query Analyzer Program
PetClinic	13
AjaxCrudJTable	5

Table 5.2: The total number of SQL queries identified by SQL Query Analyzer Program in the sample applications

These results were verified manually by checking the Data Access Layer Java classes for SQL Query Strings and found that the SQL Query Analyzer Program produced exact results. In the next section we will discuss about the evaluation of the Database Migration Tool.

5.2 Evaluation of Database Migration Tool

We applied the Database Migration Tool on our sample applications to migrate the application data present in the tables to the GAE Datastore and it successfully migrated all the tables in the Database schemas of the sample applications to GAE Remote Datastore. The Database migration Tool was configured to run both Data Exporter and Data Importer modules with a thread pool size equal to 5 and fetch size equal to 500. This configuration means the Data Exporter module uses a thread pool containing 5 worker threads to export the tables to CSV files and the Data Importer module uses a thread pool containing 5 worker threads to read the CSV files from file system to GAE Datastore. The maximum number of threads active in both thread pools in this case is 5. The threads are reused to execute any pending tasks. The fetch size 500 indicates that each worker thread in the thread pool created in the DataExporter class iteratively fetches 500 records from the table and creates a CSV file containing these 500 records. For example if a table has 2500 records, maximum 5 CSV files will be created to export the records and each CSV file can contains 500 table records. Also each worker thread in the thread pool created in the DataImporter class iteratively reads the CSV files and creates entities in GAE Datasore in batches of size 500. In addition, there will be at most 5 worker threads actively processing the CSV files in the Data Importer module.

The Table 5.3 gives a summary of the database migration performed for moving the data from the RDBMS to the GAE Datastore using the Database Migration Tool. We repeated the data migration process 3 times for both sample applications in order to take an average value.

	# Tables migrated	# Records migrated	Migration average time (ms)
PetClinic	7	49	$\frac{7059+7607+7497}{3} = 7387ms$
AjaxCrudJTable	1	4497	$\frac{95367+88809+96745}{3} = 93640ms$

Table 5.3: The total number of records and tables in the RDBMS of the sample applications migrated to the GAE Datastore by the Database Migration Tool

Both the sample applications helped us to evaluate different aspects of our Database Migration Tool. The PetClinic application has a total of 7 tables containing 49 records. The Database Migration Tool migrated the 7 tables containing 49 records to GAE Remote Datastore and took an average of 7387 milliseconds to migrate the PetClinic applicaton data. Since the PetClinic application having 7 tables enable us to test the multithreading aspects of both Database Exporter module and Data Importer module. The AjaxCrudJTable application has just one table containing 4497 records and the Database Migration Tool took an average of 93640 milliseconds to migrate the PetClinic applicaton data. This indicates that the Database Migration Tool is capable of moving hugh quantities of data from the Database management systems of the applications to GAE Datastore.

We have also evaluated the Database Migration Tool for performance. First we evaluated the performance on the PetClinic application by migrating the Petclinic application's data multiple times to the GAE Datastore by varying the thread pool size as well as the fetch size configuration properties of the Database Migration Tool. We have observed that when the thread pool size of both Data Exporter module and Data Importer module and the fetch size are increased faster the rate of data migration from the RDBMS to GAE Datastore. The Figure 5.1 clearly illustrates this. This is because when increasing the thread pool size basically makes more threads available for exporting the tables in the applicaton to GAE Datastore. Also increasing the fetch size configuration property value indicates Database Migration Tool sending a larger chunk of data to the GAE Datastore each time the Datastore is accessed via Remote APIs.

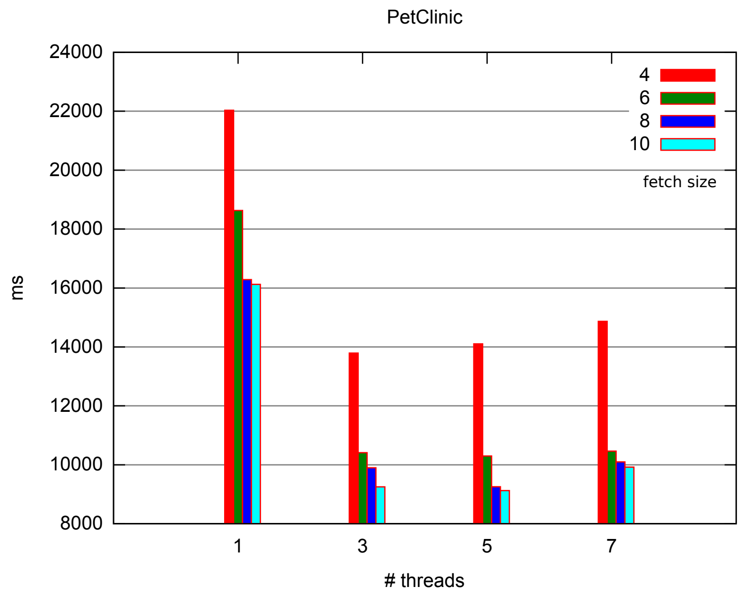


Figure 5.1: Performance evaluation of the Database Migration Tool on PetClinic

Next we evaluated the performance of the Database Migration Tool on the AjaxCrud-JTable application. The AjaxCrudJTable application has just one table and the table contains around 4497 records. Since there is only one table to be migrated thread pool size of 1 is sufficient. We have evaluated the performance of the Database Migration Tool by varying the fetch size. The following Figure 5.2 shows describes the result of the evaluation of the Database Migration Tool on varying fetch size. It can be observed that more the fetch size the faster the data migration from the RDBMS to the GAE Datastore.

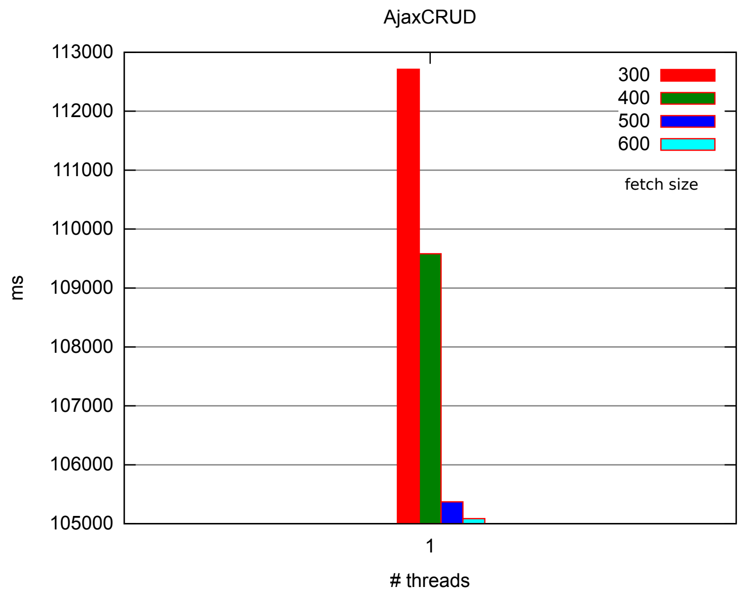


Figure 5.2: Performance evaluation of the Database Migration Tool on AjaxCrudJTable

To summarize, the Database Migration Tool achieves performance scalablity as designed.

5.3 Summary

We applied the proposed software methodology on the sample applications and we show that our approach is correct as we are able to migrate the applications to the GAE Cloud. Both the Java Source Code Analyzer Tool and Database Migration Tool played a key role in migrating the applications to the GAE Platform. The developers now can rely on the software methodology to migrate application to GAE platform easily and with less efforts. The Developers do not have to concern about installing Google Plugin in their IDEs, creating a GAE project in the workspace and copying the application files to the created GAE project, searching for GAE unsupported JRE classes and SQL queries in the application code for refactoring and using trivial bulkloader tool for database migration from the RDBMS of the application to GAE Datastore. The sample applications were easily refactored using the inputs from Java Source Code Analyzer Tool and the tables in the RDBMS used by these application were migrated using Database Migration Tool. we also found the Database Migration Tool's efficiency can be optimised by increasing the thread pool size and fetch size. Both the migrated versions of PetClinic[4] and AjaxCrudJTable[5] applications can be accessed from the Google App Engine Cloud.

[4]http://spring-petclinic.appspot.com/
[5]http://ajaxcrudjtable.appspot.com/

Chapter 6

Conclusions

Cloud Computing is the latest paradigm delivering the IT services as computing utility over internet. In this dissertation, we have proposed a software methodology that helps to efficiently migrate existing Java/JEE applications to Google App Engine Platform. The main aim of this Software methodology is to promote the adoption of GAE PaaS Cloud for the existing Java/JEE applications running on traditional corporate IT infrastructures. The applications deployed on GAE can benefits from GAE platform features. The GAE Platform provides features such as automatic scaling, load balancing and scalable data storage that supports transactions to name a few. However it can be quite challenging to migrate an existing application to the Google App Engine as it involves modifying the application source code as well as migrating the application's data to the Google App Engine Platform.

The thesis identified the need to develop a Java Source Code Analyzer Tool and a Database Migration Tool which are used as part of the Software methodology. The thesis also provided the designs for both Java Source Code Analyzer Tool and Database Migration Tool. The tools were implemented as per the design and worked as expected. The Software methodology was evaluated with the aid of these tools developed by migrating sample applications to the GAE platform. The research has been very positive as the tools developed provided accurate results. The Java Source Code Analyzer helped to identify all the GAE unsupported JRE classes and the usage of SQL queries in the applications there by aiding to refactor the applications effortlessly. It is also noticed that the standard available tools such as Google Plugin for eclipse has limited runtime code validation support. The Database Migration Tool was able to migrate entire the application data from the RDBMS to the GAE Datastore. The Database Migration Tool proved to be efficient with the use of multithreading technology as well as can handle the migration of hugh quantities of data without any failures. Another important observation is the maturity of the Google App Engine Platform. The Google App Engine Platform provides a lot of noteworthy features and one of them is the Remote feature which enables external applications to access the GAE services. The Remote API's is used by the Database Migration Tool to access the production Datastore thereby enabling Data migration. Hence to conclude, It is possible to efficiently migrate a Java/JEE prototype application to the Google App Engine PaaS Cloud.

6.1 Further work

Currently the Google App Engine BlackList Analyzer program of Java Source Code Analyzer Tool can find the presence of GAE unsupported JRE classes only in the application's Java source files. The Google App Engine BlackList Analyzer Program should be modified in the future so that it can scan for the presence of GAE unsupported JRE classes in the library files(JAR files) used by the application. Further, the Google App Engine BlackList Analyzer Program could be extended for validating various Java Frameworks and third party Java libraries for their compatibility with GAE Platform. In the future the Google App Engine BlackList Analyzer Program should be modified so that it should be able to scan for Java class files containing Java byte code.

Currently the Java Source Code Analyzer Tool is used by copying the Java Source Code Analyzer Tool files into a Java project's root folder, manually editing the configuration properties files and invoking the interfaces provided by the GAE BlackList Analyzer Program and SQL Query Analyzer Program. However, this is slightly an inconvinient approach. The best approach will be bundling the Java files of the Java Source Code Analyzer Tool as a Plugin, so that this Plugin can be installed on variety of Integrated Development Environments(IDE) such as Eclipse, NetBeans etc. The users can simply install the Java Source Code Analyzer Tool as Plugin on an IDE and can scan for the presence of GAE unsupported JRE classess or SQL queries present in the application's Java files using graphical user interfaces provided by the IDE.

Equally important, the Database Migration Tool should be tested with various Database systems from different vendors and also should be made sure that it can migrate millions of records from Database to GAE Datastore. The tool could also be improved to have a fault tolerant design so that the migration process could still continue in case if a failure occurs during the execution.

Bibliography

Angabini, A., Yazdani, N., Mundt, T. & Hassani, F. (2011), Suitability of cloud computing for scientific data analyzing applications; an empirical study, *in* F. Xhafa, L. Barolli, J. Kolodziej & S. U. Khan, eds, '*2011 International Conference on P2P, Parallel,Grid, Cloud and Internet Computing*', IEEE, Barcelona, Catalonia, Spain, pp. 193–199.

Bhat, U. & Jadhav, S. (2010), 'Moving towards non-relational databases', *International Journal of Computer Applications* **1**(13), 40–46. Published By Foundation of Computer Science.

Bonnet, L., Laurent, A., Sala, M., Laurent, B. & Sicard, N. (2011), Reduce, you say: What nosql can do for data aggregation and bi in large repositories, *in* '*Proceedings of the 2011 22Nd International Workshop on Database and Expert Systems Applications*', IEEE, Toulouse, France, pp. 483–488.

Bunch, C., Chohan, N., Krintz, C., Chohan, J., Kupferman, J., Lakhina, P., Li, Y. & Nomura, Y. (2010), An evaluation of distributed datastores using the appscale cloud platform, *in* '*Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*', IEEE, Miami, Florida, USA, pp. 305–312.

Buyya, R., Vechhiola, C. & Selvi, S.T (2013), *Mastering Cloud Computing Technologies and Applications Programming*, Morgan Kaufmann.

Chauhan, M. & Babar, M. (2012), Towards process support for migrating applications to cloud computing, *in* '*2012 International Conference on Cloud and Service Computing (CSC)*', IEEE, Shanghai, China, pp. 80–87.

Da-sheng, W. & Sheng-yu, W. (2010), Dynamically maintain the teaching examples of triggers and stored procedures about the course of database application, *in* '2010 2nd International Conference on Education Technology and Computer (ICETC)', Vol. 1, IEEE, Shanghai, China, pp. 25–27.

Dewan, H. & Hansdah, R. C. (2011), A survey of cloud storage facilities, *in* '2011 IEEE World Congress on Services', IEEE, Washington, DC, USA, pp. 224–231.

Google Developers (2013a), 'Blobstore java api overview', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/java/blobstore/. [Accessed 6 December 2013].

Google Developers (2013b), 'Java datastore api', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/java/datastore/. [Accessed 6 December 2013].

Google Developers (2013d), 'The jre class white list', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/java/jrewhitelist. [Accessed 6 December 2013].

Google Developers (2013e), 'Transactions', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/python/datastore/transactions. [Accessed 6 December 2013].

Google Developers (2013f), 'Uploading and downloading data', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/python/tools/uploadingdata. [Accessed 6 December 2013].

Google Developers (2013g), 'Java runtime environment - java google developers', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/java/. [Accessed 23 July 2014].

Google Developers (2013h), 'Java google developers', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/java/datastore/transactions. [Accessed 23 July 2014].

Google Developers (2013j), 'Google cloud datastore google developers', [Online] *Google Developers*. Available from: https://developers.google.com/datastore/docs/concepts/gql. [Accessed 09 August 2014].

Google Developers (2014k), 'Java google developers', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/docs/java/datastore/queries. [Accessed 19 August 2014].

Gregorio, J. (2008), 'Sharding counters', [Online] *Google Developers*. Available from: https://developers.google.com/appengine/articles/sharding_counters?hl=en. [Accessed 14th December 2013].

Gu, H., Diao, Y., Liu, W. & Zhang, X. (2011), The design of smart home platform based on cloud computing, *in* '2011 International Conference on Electronic and Mechanical Engineering and Information Technology (EMEIT)', Vol. 8, IEEE, Harbin, China, pp. 3919–3922.

Hexiao, H., Shiming, Z. & Haijian, C. (2012), 'Reengineering from tradition to cloud: A case study', *Procedia Engineering* **29**(0), 2638 – 2643. 2012 International Workshop on Information and Electronics Engineering.

Kotecha, S., Bhise, M. & Chaudhary, S. (2011), Query translation for cloud databases, *in* '*2011 Nirma University International Conference on Engineering Current Trends in Technology*', IEEE, Ahmedabad, Gujarat, India, pp. 1–4.

Lei Hu, Peng Yue, H. Z. (2012), Geoprocessing in google cloud computing: Case studies, *in* 'The First International Conference on Agro-Geoinformatics', IEEE, Shanghai, pp. 1–6.

Li, J., Li, X., Liu, G. & He, Z. (2010), Log management approach in three-dimensional spatial data management system, *in* 'Geoinformatics, 2010 18th International Conference on', IEEE, Beijing, China, pp. 1–5.

Malawski, M., Kuzniar, M., Wojcik, P. & Bubak, M. (2013), 'How to use google app engine for free computing', *Internet Computing, IEEE* **17**(1), 50–59.

Martins, J. (2012), 'Dealing with deadlineexceedederrors', [Online] *Google Developers*. Available from:

https://developers.google.com/appengine/articles/deadlineexceedederrors. [Accessed 6th December 2013].

Mell, P. & Grance, T. (2011), 'The NIST Definition of Cloud Computing', [Online].*National Institute of Standards and Technology* .Available from: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf [Accessed 12th December 2013].

Oracle (2011), 'Java se 6 features and enhancements', [Online] *Google Developers*. Available from: http://docs.oracle.com/javase/6/docs/api/javax/tools/JavaCompiler.html. [Accessed 01 August 2014].

Prodan, R. & Sperk, M. (2013), 'Scientific computing with google app engine', *Future Generation Computer Systems* **29**(7), 1851 – 1859.

Ramanathan, S., Goel, S. & Alagumalai, S. (2011), Comparison of cloud database: Amazon's simpledb and google's bigtable, *in* '*2011 International Conference on Recent Trends in Information Systems (ReTIS)*', IEEE, Jadavpur University,Kolkata, India, pp. 165–168.

Sanderson, D. (2009), *Programming Google App Engine*, OŔeilly Media.

Shu-Qing, Z. & Jie-Bin, X. (2010), The improvement of paas platform, *in* '*Proceedings of the 2010 First International Conference on Networking and Distributed Computing*', IEEE, Hangzhou,Zhejian,China, pp. 156–159.

Srirama, S. N., Jakovits, P. & Vainikko, E. (2012), 'Adapting scientific computing problems to clouds using mapreduce', *Future Generation Computer Systems* **28**(1), 184 – 192.

Tran, V., Keung, J., Liu, A. & Fekete, A. (2011), Application migration to cloud: A taxonomy of critical factors, *in* 'Proceedings of the 2Nd International Workshop on Software Engineering for Cloud Computing', SECLOUD '11, ACM, New York, NY, USA, pp. 22–28.

Vu, Q. H. & Asal, R. (2012), Legacy application migration to the cloud: Practicability and methodology, *in* '*2012 IEEE Eighth World Congress on Services*', IEEE, Honolulu, Hawaii, USA, pp. 270–277.

Yin, H., Han, J., Liu, J. & Dong, J. (2011), The application research of gae on e-learning - taking google cloudcourse for example, *in* '*2011 IEEE 3rd International Conference on Communication Software and Networks (ICCSN)*', IEEE, Xi'an, China, pp. 156–159.

Zahariev, A. (2009), 'Google app engine', *Helsinki University of Technology* .

Appendix A

Appendix

A.1 Java Source Code Analyzer Tool important Java classes and configuration files

A.1.1 GAE BlackList Analyzer Program important Java classes

```java
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.annotation.processing.AbstractProcessor;
import javax.tools.JavaCompiler;
import javax.tools.JavaCompiler.CompilationTask;
import javax.tools.JavaFileObject;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;

/**
 * The main class for starting the GAE BlackList Analyzer Program.
 *
 * @author Prasanth M P
 *
 */
public class GAEBlackListAnalyzerMain {

  public static void main(String[] args) throws IOException {
    // Gets the Java programming language compiler
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    // Get a new instance of the standard file manager implementation
    StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null
        , null);
    // Get the valid source files as a list
    List<File> files = new ArrayList<File>();
    List<String> optionList = new ArrayList<String>();
    optionList.addAll(Arrays.asList(System.getProperty("java.class.path")));
    files.addAll(FileSearchMain.getProjectFiles(ConfigurationPropertiesFileReader.
        getProperty("project.rootfolder.location")));
    if (files.size() > 0) {
      // Get the list of java file objects
      Iterable<? extends JavaFileObject> compilationUnits = fileManager.
          getJavaFileObjectsFromFiles(files);
      // Create the compilation task
      CompilationTask task = compiler.getTask(null, fileManager, null,null, null,
          compilationUnits);
      // Get the list of annotation processors
      LinkedList<AbstractProcessor> processors = new LinkedList<AbstractProcessor>()
          ;
      processors.add(new BlackListCodeAnalyzerProcessor());
      task.setProcessors(processors);
      // Perform the compilation task.
      task.call();
      try {
        fileManager.close();
      } catch (IOException e) {
        System.out.println(e.getLocalizedMessage());
      }
    } else {
      System.out.println("No valid source files to process. Extiting from the
          program");
      System.exit(0);
    }
  }

}
```

Listing A.1: The GAEBlackListAnalyzerMain.java class

```
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;

```

```java
import com.sun.source.util.TreePath;
import com.sun.source.util.Trees;

/**
 * The annotation processor class for the GAE BlackList Analyzer Program.
 *
 * @author Prasanth M P.
 */
@SupportedAnnotationTypes("*")
public class BlackListCodeAnalyzerProcessor extends AbstractProcessor {

  private Trees tree;

  @Override
  public synchronized void init(ProcessingEnvironment processingEnv) {
    super.init(processingEnv);
    tree = Trees.instance(processingEnv);
  }

  /**
   * The compiler invokes the process method for processing annotations in a
   * series of rounds. The annotation processing tool framework will provide
   * an annotation processor with an object implementing ProcessingEnvironment
   * interface so the processor can use facilities provided by the framework
   * to write new files, report error messages, and find other utilities. The
   * object implementing ProcessingEnvironment interface is then passed to the
   * Visitor object of class BlackListCodeAnalyzerVisitor via constructor. The
   * BlackListCodeAnalyzerVisitor can use the object implementing
   * ProcessingEnvironment to report error messages.
   */
  @Override
  public boolean process(Set<? extends TypeElement> annotations,
      RoundEnvironment roundEnv) {
    BlackListCodeAnalyzerVisitor visitor = new BlackListCodeAnalyzerVisitor(
        processingEnv);
    // returns the Java classes, interfaces and packages in the compilation unit as ↩
        a Set containing Element objects.
    Set<? extends Element> elements = roundEnv.getRootElements();
    for (Element element : elements) {
      //Gets the TreePath node for a given Element. The TreePath consist of a tree ↩
          of nodes(Class nodes, method nodes, variable nodes etc)
      TreePath treePath = tree.getPath(element);
      if (treePath != null) {
        visitor.scan(treePath, tree);
      }
    }
    return false;
  }
}
```

Listing A.2: The BlackListCodeAnalyzerProcessor.java class

```
import java.net.URLClassLoader;
import java.util.List;

import javax.annotation.processing.ProcessingEnvironment;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;
import javax.lang.model.type.TypeMirror;

import com.sun.source.tree.ClassTree;
import com.sun.source.tree.VariableTree;
import com.sun.source.util.SourcePositions;
import com.sun.source.util.TreePathScanner;
import com.sun.source.util.Trees;
import com.sun.tools.javac.api.JavacTrees;
import com.sun.tools.javac.code.Symbol.VarSymbol;

/**
 * The visitor object implementation that visits the class and method nodes in
 * the Abstract Syntax Tree(AST) representation of a Java source file. A java
 * source file can be represented as a tree of nodes and each node represent a
 * meaninful construct.
 *
 * @author Prasanth M P
 */
public class BlackListCodeAnalyzerVisitor extends
    TreePathScanner<Object, Object> {

  private ProcessingEnvironment processingEnv;

  private URLClassLoader urlClassLoader;

  public BlackListCodeAnalyzerVisitor(ProcessingEnvironment processingEnv) {
    this.processingEnv = processingEnv;
    this.urlClassLoader = AppEngineAnalyzerUtils.getUrlClassLoader( ↩
        ConfigurationPropertiesFileReader.getProperty("project.rootfolder.location") ↩
        );
  }

  /**
   * The visitor method is invoke when a class declaration is found in the
   * Java file. Within the method it invoke methods that check if the class
   * extends or implements an GAE unsupported JRE Class.
   */
```

```
@Override
public Object visitClass(ClassTree node, Object arg1) {
  Trees trees = (Trees) arg1;
  Element element = trees.getElement(getCurrentPath());
  if (AppEngineAnalyzerUtils.isClass(element)) {
    TypeElement clazz = (TypeElement) element;
    // check for JRE super class which is blacklisted
    checkIfSuperClassBlackListed(clazz, urlClassLoader);
    // check if it implements restricted interfaces
    checkIfInterfaceBlackListed(clazz, urlClassLoader);
    // check if it has restricted field variables
  }
  return super.visitClass(node, arg1);
}

/**
 * The visitor method is invoked when a variable declaration is found in the
 * Java class declared as a field variable or method variable. The method checks
 * if the variable type is GAE unsupported JRE class.
 */
@Override
public Object visitVariable(VariableTree node, Object p) {
  try {
    JavacTrees javacTrees = (JavacTrees) p;
    Element element = javacTrees.getElement(getCurrentPath());
    VarSymbol s = (VarSymbol) element;
    TreeUtils.className(javacTrees.getElement(getCurrentPath()));
    SourcePositions sourcePositions = javacTrees.getSourcePositions();
    long startPosition = sourcePositions.getStartPosition(
        getCurrentPath().getCompilationUnit(), node);
    javacTrees.getTree(element);
    Class declaredType = urlClassLoader
        .loadClass(s.asType().toString());
    for (Class blackListedClazz : GoogleAppEngineBlackList
        .getBlackList()) {
      if (blackListedClazz.isAssignableFrom(declaredType)) {
        processingEnv
          .getMessager()
          .printMessage(
              javax.tools.Diagnostic.Kind.ERROR,
              "The Class "
                  + TreeUtils.className(javacTrees
                      .getElement(getCurrentPath()))
                  + "  has a declared type which is an app engine restricted JRE ↩
                      Interface/class "
                  + declaredType
                  + " at line number "
                  + getCurrentPath()
                      .getCompilationUnit()
                      .getLineMap()
                      .getLineNumber(
                          startPosition)
                  + "\n Rule: "
                  + GoogleAppEngineBlackList
                      .getRule(declaredType));
      }
    }
    // }
  } catch (Exception e) {
  }
  return super.visitVariable(node, p);
}

/**
 * The method has the logic to check if a Java class implements an interface class ↩
     which is a GAE unsupported JRE class.
 */
private void checkIfInterfaceBlackListed(TypeElement clazz,
    URLClassLoader urlClassLoader) {
  List<? extends TypeMirror> interfaces = clazz.getInterfaces();
  for (TypeMirror typeMirror : interfaces) {
    try {
      Class interfaceClazz = urlClassLoader.loadClass(typeMirror
          .toString());
      for (Class blackListedClazz : GoogleAppEngineBlackList
          .getBlackList()) {
        if (blackListedClazz.isAssignableFrom(interfaceClazz)) {
          processingEnv
            .getMessager()
            .printMessage(
                javax.tools.Diagnostic.Kind.ERROR,
                "The Class "
                    + clazz
                    + " implements an app engine restricted JRE Interface "
                    + interfaceClazz
                    + "\n Rule: "
                    + GoogleAppEngineBlackList
                        .getRule(interfaceClazz));
        }
      }
    } catch (ClassNotFoundException e) {
    }
  }
}
```

```

  /**
   * The method has the logic to check if a Java class implements an JRE class which is a GAE unsupported JRE class.
   */
  private void checkIfSuperClassBlackListed(TypeElement clazz,
      URLClassLoader urlClassLoader) {
    try {
      Class elementSuperClass = urlClassLoader.loadClass(clazz
          .getSuperclass().toString());
      for (Class blackListedClass : GoogleAppEngineBlackList
          .getBlackList()) {
        if (blackListedClass.isAssignableFrom(elementSuperClass)) {
          processingEnv
            .getMessager()
            .printMessage(
                javax.tools.Diagnostic.Kind.ERROR,
                "The Class "
                    + clazz
                    + " extends an app engine restricted JRE Class "
                    + blackListedClass
                    + "\n Rule: "
                    + GoogleAppEngineBlackList
                        .getRule(elementSuperClass));
          return;
        }
      }

    } catch (ClassNotFoundException e) {}
  }
}
```

Listing A.3: The BlackListCodeAnalyzerVisitor.java class

```
error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app engine restricted JRE Interface/class interface java.sql.Connection at line number 16 Rule: Google App Engine will not support JDBC Connection class.
Please use Datastore API's to get a connection to Datastore [DatastoreService datastore = DatastoreServiceFactory.getDatastoreService()]

error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app engine restricted JRE Interface/class interface java.sql.PreparedStatement at line number 25 Rule: Google App Engine will not support JDBC PreparedStatement.class API's.
Please use Datastore API's to perform CRUD Operation

```

```
error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app ↩
      engine restricted JRE Interface/class interface java.sql.PreparedStatement at  ↩
     line number 41 Rule: Google App Engine will not support JDBC PreparedStatement. ↩
     class API's.
Please use Datastore API's to perform CRUD Operation

error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app ↩
      engine restricted JRE Interface/class interface java.sql.PreparedStatement at  ↩
     line number 53
Rule: Google App Engine will not support JDBC PreparedStatement.class API's.
Please use Datastore API's to perform CRUD Operation

error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app ↩
      engine restricted JRE Interface/class interface java.sql.Statement at line  ↩
     number 70 Rule: Google App Engine will not support JDBC Statement.class API's.
Please use Datastore API's to perform CRUD Operation

error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app ↩
      engine restricted JRE Interface/class interface java.sql.ResultSet at line  ↩
     number 71 Rule: Google App Engine will not support JDBC ResultSet.class API's.
Please use Datastore API's

error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app ↩
      engine restricted JRE Interface/class interface java.sql.PreparedStatement at  ↩
     line number 90 Rule: Google App Engine will not support JDBC PreparedStatement. ↩
     class API's.
Please use Datastore API's to perform CRUD Operation

error: The Class com.programmingfree.dao.CrudDao has a declared type which is an app ↩
      engine restricted JRE Interface/class interface java.sql.ResultSet at line  ↩
     number 93 Rule: Google App Engine will not support JDBC ResultSet.class API's.
Please use Datastore API's

error: The Class com.programmingfree.utility.DBUtility has a declared type which is  ↩
     an app engine restricted JRE Interface/class interface java.sql.Connection at  ↩
     line number 13 Rule: Google App Engine will not support JDBC Connection class.
Please use Datastore API's to get a connection to Datastore [DatastoreService  ↩
     datastore = DatastoreServiceFactory.getDatastoreService()]
```

Listing A.4: The error logs generated by the GAE BlackList Analyzer Program for the AjaxCrudJTable application.

A.1.2 SQL Query Analyzer Program important Java classes

```
import java.io.File;
import java.io.IOException;
```

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;
import java.util.List;

import javax.annotation.processing.AbstractProcessor;
import javax.tools.JavaCompiler;
import javax.tools.JavaCompiler.CompilationTask;
import javax.tools.JavaFileObject;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;

/**
 * The main class for starting the SQL Query Analyzer Program.
 *
 * @author Prasanth M P
 *
 */
public class SQLQueryAnalyzerMain {

  public static void main(String[] args) {

    // Gets the Java programming language compiler
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    // Get a new instance of the standard file manager implementation
    StandardJavaFileManager fileManager = compiler.getStandardFileManager(null, null ↩
        , null);
    List<File> files = new ArrayList<File>();
    List<String> optionList = new ArrayList<String>();
    optionList.addAll(Arrays.asList(System.getProperty("java.class.path")));
    files.addAll(FileSearchMain.getProjectFiles(ConfigurationPropertiesFileReader. ↩
        getProperty("project.jdbc.files.location")));
    SQLRuleRepository.showRules();
    if (files.size() > 0) {
      // Get the list of java file objects
      Iterable<? extends JavaFileObject> compilationUnits = fileManager. ↩
          getJavaFileObjectsFromFiles(files);
      // Create the compilation task
      CompilationTask task = compiler.getTask(null, fileManager, null, null, null,  ↩
          compilationUnits);
      // Get the list of annotation processors
      LinkedList<AbstractProcessor> processors = new LinkedList<AbstractProcessor>() ↩
          ;
      processors.add(new JdbcDAOCodeAnalyzerProcessor());
      task.setProcessors(processors);
      // Perform the compilation task.
      task.call();
      try {
        fileManager.close();
      } catch (IOException e) {
        System.out.println(e.getLocalizedMessage());
      }
    } else {
      System.out.println("No valid source files to process. "
          + "Extiting from the program");
      System.exit(0);
    }
  }

}
```

Listing A.5: The SQLQueryAnalyzerMain.java class

```
import java.util.Set;

import javax.annotation.processing.AbstractProcessor;
import javax.annotation.processing.ProcessingEnvironment;
import javax.annotation.processing.RoundEnvironment;
import javax.annotation.processing.SupportedAnnotationTypes;
import javax.lang.model.element.Element;
import javax.lang.model.element.TypeElement;

import com.sun.source.util.TreePath;
import com.sun.source.util.Trees;

/**
 * The annotation processor class for the SQL Query Analyzer Program.
 *
 * @author Prasanth M P
 *
 */
@SupportedAnnotationTypes("*")
public class JdbcDAOCodeAnalyzerProcessor extends AbstractProcessor {

  private Trees tree;

  @Override
  public synchronized void init(ProcessingEnvironment processingEnv) {
    super.init(processingEnv);
    tree = Trees.instance(processingEnv);
  }

  /**
   * The compiler invokes the process method for processing annotations in a
   * series of rounds. The annotation processing tool framework will provide
   * an annotation processor with an object implementing ProcessingEnvironment
   * interface so the processor can use facilities provided by the framework
   * to write new files, report error messages, and find other utilities. The
   * object implementing ProcessingEnvironment interface is then passed to the
   * Visitor object of class JdbcCodeAnalyzerVisitor via constructor. The
   * JdbcCodeAnalyzerVisitor can use the object implementing
   * ProcessingEnvironment to report error messages.
   */
  @Override
  public boolean process(Set<? extends TypeElement> annotations, RoundEnvironment ↩
      roundEnv) {
    JdbcCodeAnalyzerVisitor visitor = new JdbcCodeAnalyzerVisitor( processingEnv);
    // returns the Java classes, interfaces and packages in the compilation unit as ↩
        a Set containing Element objects.
    Set<? extends Element> elements = roundEnv.getRootElements();
    for (Element element : elements) {
      //Gets the TreePath node for a given Element. The TreePath consist of a tree ↩
          of nodes(Class nodes, method nodes, variable nodes etc).
      TreePath treePath = tree.getPath(element);
          visitor.scan(treePath, tree);
    }
    return false;
  }
}
```

Listing A.6: The JdbcDAOCodeAnalyzerProcessor.java class

```
import java.util.List;

import javax.annotation.processing.ProcessingEnvironment;

import com.sun.source.tree.BlockTree;
import com.sun.source.tree.CompilationUnitTree;
import com.sun.source.tree.EnhancedForLoopTree;
import com.sun.source.tree.ExpressionTree;
import com.sun.source.tree.IfTree;
import com.sun.source.tree.MethodInvocationTree;
import com.sun.source.tree.MethodTree;
import com.sun.source.tree.ReturnTree;
import com.sun.source.tree.StatementTree;
import com.sun.source.tree.Tree.Kind;
import com.sun.source.tree.TryTree;
import com.sun.source.tree.VariableTree;
import com.sun.source.util.SourcePositions;
import com.sun.source.util.TreePathScanner;
import com.sun.source.util.Trees;

/**
```

```
 * The visitor object implementation that visits the method nodes in the Abstract Syntax Tree(AST) representation of a Java source file.
 * A java source file can be represented as a tree of nodes and each node represent a meaninful construct.
 *
 * @author Prasanth M P
 *
 */
public class JdbcCodeAnalyzerVisitor extends TreePathScanner<Object, Object> {

  private ProcessingEnvironment processingEnv;

  public JdbcCodeAnalyzerVisitor(ProcessingEnvironment processingEnv) {
    this.processingEnv = processingEnv;
  }

  /**
   * The visitor method invoked when a method declaration is found in the Abstract syntax tree notation of the java source file.
   */
  @Override
  public Object visitMethod(MethodTree methodTree, Object arg1) {
    Trees trees = (Trees) arg1;
    CompilationUnitTree compilationUnitTree = getCurrentPath().getCompilationUnit();
    // compileTree.getSourceFile().getName(); // filename including path
    List<? extends StatementTree> statements = methodTree.getBody()
        .getStatements();
    processStatements(statements,trees,compilationUnitTree,methodTree);
    return super.visitMethod(methodTree, arg1);
  }

  /**
   * This method is recursively called to process the occurence of an SQL query string within a method.
   *
   * @param statements - The List of StatementTree nodes. The StatementTree node represent
   * a statement in the method such as if, else, else if, try, catch, for, finally etc.
   * @param trees - Provides utilities for operations on Abstract Syntax Trees.
   * @param compileTree - Represents the abstract syntax tree for compilation units (source files).
   * @param methodTree - Represents a method node within the AST representation of the Java source file.
   */
  private void processStatements(List<? extends StatementTree> statements, Trees trees,CompilationUnitTree compileTree,
      MethodTree methodTree ) {

    for (StatementTree statement : statements) {
      if (statement.getKind().equals(Kind.EXPRESSION_STATEMENT)) {
        handleMessageForExpressionStatment(trees,statement,compileTree,methodTree);
      } else if (statement.getKind().equals(Kind.VARIABLE)) {
        handleMessageForVariable(trees,statement,compileTree,methodTree);
      } else if(statement.getKind().equals(Kind.TRY) ){
        handleMessageForTryStatement(trees,statement,compileTree,methodTree);
      } else if ( statement.getKind().equals(Kind.IF)){
        handleMessageForIFStatement(trees,statement,compileTree,methodTree);
      } else if(statement.getKind().equals(Kind.RETURN)){
        handleMessageForReturnStatement(trees,statement,compileTree,methodTree);
      } else if(statement.getKind().equals(Kind.ENHANCED_FOR_LOOP)){
        handleMessageForEnhancedForLoop(trees,statement,compileTree,methodTree);
      }
    }
  }

  /**
   * The method is called when a for loop in encountered in the source file. The ↩
       method then obtains the statements within the for loop
   * and recursively calls the processStatements method.
   */
  private void handleMessageForEnhancedForLoop(Trees trees,
      StatementTree statement, CompilationUnitTree compileTree,
      MethodTree methodTree) {
    EnhancedForLoopTree enhanceForLoop = ( EnhancedForLoopTree)statement;
      enhanceForLoop.getExpression();
      List<? extends StatementTree> statements = ((BlockTree)enhanceForLoop. ↩
          getStatement()).getStatements();
    processStatements(statements, trees, compileTree, methodTree);
  }

  /**
   * The method is called when a return statement is encountered in the source file. ↩
        The method then obtains the statements within the return statement
   * and calls the handleMessageForExpressionStatment to display an error log to ↩
       indicate if String variable returned is an SQL query.
   */
  private void handleMessageForReturnStatement(Trees trees,
      StatementTree statement, CompilationUnitTree compileTree,
      MethodTree methodTree) {
    ReturnTree rt = ( ReturnTree)statement;
    if( rt.getExpression().getKind().equals(Kind.METHOD_INVOCATION) ) {
      MethodInvocationTree methodInvocationTree = (MethodInvocationTree)rt. ↩
          getExpression();
      List<? extends ExpressionTree> args = methodInvocationTree.getArguments();
      for(ExpressionTree ex: args){
        if(ex.getKind().equals(Kind.STRING_LITERAL)){
          handleMessageForExpressionStatment(trees,statement, compileTree, methodTree
              );
        }
      }
    }
  }

  /**
   * The method is called when a if else is encountered in the source file. The
       method then obtains the statements within the if else
   * and calls the processStatements recursively.
   */
  private void handleMessageForIFStatement(Trees trees,
      StatementTree statement, CompilationUnitTree compileTree,
      MethodTree methodTree) {
    IfTree iftree = (IfTree)statement;
    BlockTree ifThenBlock= (BlockTree)iftree.getThenStatement();
    ifThenBlock.getStatements();
    processStatements(ifThenBlock.getStatements(), trees, compileTree, methodTree);
    BlockTree ifElseBlock = (BlockTree)iftree.getElseStatement();
    processStatements(ifElseBlock.getStatements(), trees, compileTree, methodTree);
  }

  /**
   * The method is called when a Try block is encountered in the source file. The
       method then obtains the statements to check if the statements within
   * the Try block is of expression statements or variables and then calls
       appropriate methods for processing further.
   */
  private void handleMessageForTryStatement(Trees trees,
      StatementTree statement, CompilationUnitTree compileTree,
      MethodTree methodTree) {
    TryTree tree = (TryTree)statement;
    List<? extends StatementTree> blockStatements = tree.getBlock().getStatements();
    for(StatementTree stmt : blockStatements){
      if (stmt.getKind().equals(Kind.EXPRESSION_STATEMENT)) {
        handleMessageForExpressionStatment(trees,stmt,compileTree,methodTree);
      } else if (stmt.getKind().equals(Kind.VARIABLE)) {
        handleMessageForVariable(trees,stmt,compileTree,methodTree);
      }
    }
  }

  /**
   * checks if the string values "select ", "update ", "insert ", or "delete "  ↩
       present in the declared variable.
   */
  private void handleMessageForVariable(Trees trees, StatementTree statement,
      CompilationUnitTree compileTree, MethodTree methodTree) {
    VariableTree queryVariable = (VariableTree) statement;
    SourcePositions sourcePosition = trees.getSourcePositions();
    long startPosition = sourcePosition.getStartPosition(
        getCurrentPath().getCompilationUnit(), statement);
    if (statement.toString() != null && !statement.toString().isEmpty()) {
      if (statement.toString().toLowerCase().contains("select ")) {
        processingEnv
            .getMessager()
            .printMessage(
                javax.tools.Diagnostic.Kind.ERROR,
                "Use  Data Store retrieve entity operations on line "
                    + compileTree.getLineMap()
                        .getLineNumber(
                            startPosition)
                    + " for the jdbc query string variable "
                    + queryVariable.getName()
                    + " in the method "
                    + methodTree.getName()
                    + " in the class "
                    + TreeUtils.className(trees
                        .getElement(getCurrentPath()))
                    + " [ "
                    + compileTree.getSourceFile()
                        .getName() + " ] "+" Refer Rules "+ SQLRuleRepository.getRule( ↩
                            Operation.READ) + " in the Rule Table");
      } else if (statement.toString().toLowerCase()
          .contains("update ")) {
        processingEnv
            .getMessager()
            .printMessage(
                javax.tools.Diagnostic.Kind.ERROR,
                "Use  Data Store update entity operations on line "
                    + compileTree.getLineMap()
                        .getLineNumber(
                            startPosition)
                    + " for the jdbc query string variable "
                    + queryVariable.getName()
                    + " in the method "
                    + methodTree.getName()
                    + " in the class "
                    + TreeUtils.className(trees
                        .getElement(getCurrentPath()))
```

```
                    + " [ "
                    + compileTree.getSourceFile()
                        .getName() + " ] "+" Refer Rules "+ SQLRuleRepository.getRule( ↩
                            Operation.UPDATE) + " in the Rule Table");
      } else if (statement.toString().toLowerCase()
          .contains("insert ")) {
        processingEnv
            .getMessager()
            .printMessage(
                javax.tools.Diagnostic.Kind.ERROR,
                "Use  Data Store create entity operations on line "
                    + compileTree.getLineMap()
                        .getLineNumber(
                            startPosition)
                    + " for the jdbc query string variable "
                    + queryVariable.getName()
                    + " in the method "
                    + methodTree.getName()
                    + " in the class "
                    + TreeUtils.className(trees
                        .getElement(getCurrentPath()))
                    + " [ "
                    + compileTree.getSourceFile()
                        .getName() + " ] "+" Refer Rules "+ SQLRuleRepository.getRule( ↩
                            Operation.CREATE) + " in the Rule Table");
      } else if (statement.toString().toLowerCase()
          .contains("delete ")) {
        processingEnv
            .getMessager()
            .printMessage(
                javax.tools.Diagnostic.Kind.ERROR,
                "Use  Data Store delete entity operations on line "
                    + compileTree.getLineMap()
                        .getLineNumber(
                            startPosition)
                    + " for the jdbc query string variable "
                    + queryVariable.getName()
                    + " in the method "
                    + methodTree.getName()
                    + " in the class "
                    + TreeUtils.className(trees
                        .getElement(getCurrentPath()))
                    + " [ "
                    + compileTree.getSourceFile()
                        .getName() + " ] "+" Refer Rules "+ SQLRuleRepository.getRule( ↩
                            Operation.DELETE) + " in the Rule Table");
      }
```

```
    }
  }

  /**
   * checks if there is a string "select ", "update ", "insert ", or "delete " in ↩
       the statement.
   */
  private void handleMessageForExpressionStatment(Trees trees,StatementTree stmt, ↩
      CompilationUnitTree compilationUnitTree, MethodTree methodTree) {
    SourcePositions sourcePosition = trees.getSourcePositions();
    long startPosition = sourcePosition.getStartPosition(
        getCurrentPath().getCompilationUnit(), stmt);
    if (stmt.toString() != null && !stmt.toString().isEmpty()) {
      if (stmt.toString().toLowerCase().contains("select ")) {
        processingEnv.getMessager().printMessage(
            javax.tools.Diagnostic.Kind.ERROR,
            "Use Data Store retrieve entity operations on line "
                + compilationUnitTree.getLineMap()
                    .getLineNumber(startPosition)
                + " in the method "
                + methodTree.getName().toString()
                + " in the class "
                + TreeUtils.className(trees
                    .getElement(getCurrentPath()))
                + " ["
                + compilationUnitTree.getSourceFile().getName()
                + "] "+" Refer Rules "+ SQLRuleRepository.getRule(Operation.READD) + " ↩
                    in the Rule Table");
      } else if (stmt.toString().toLowerCase()
          .contains("update ")) {
        processingEnv.getMessager().printMessage(
            javax.tools.Diagnostic.Kind.ERROR,
            "Use Data Store update entity operations on line "
                + compilationUnitTree.getLineMap()
                    .getLineNumber(startPosition)
                + " in the method "
                + methodTree.getName().toString()
                + " in the class "
                + TreeUtils.className(trees
                    .getElement(getCurrentPath()))
                + " ["
                + compilationUnitTree.getSourceFile().getName()
                + "] "+" Refer Rules "+ SQLRuleRepository.getRule(Operation.UPDATE) + ↩
                    " in the Rule Table");
      } else if (stmt.toString().toLowerCase()
          .contains("insert ")) {
        processingEnv.getMessager().printMessage(
            javax.tools.Diagnostic.Kind.ERROR,
            "Use  Data Store create entity operations on line "
                + compilationUnitTree.getLineMap()
                    .getLineNumber(startPosition)
                + "  in the method "
                + methodTree.getName().toString()
                + " in the class "
                + TreeUtils.className(trees
                    .getElement(getCurrentPath()))
                + " ["
                + compilationUnitTree.getSourceFile().getName()
                + "] "+" Refer Rules "+ SQLRuleRepository.getRule(Operation.CREATE) + ↩
                     " in the Rule Table");
      } else if (stmt.toString().toLowerCase()
          .contains("delete ")) {
        processingEnv.getMessager().printMessage(
            javax.tools.Diagnostic.Kind.ERROR,
            "Use  Data Store delete entity operations on line "
                + compilationUnitTree.getLineMap()
                    .getLineNumber(startPosition)
                + "  in the method "
                + methodTree.getName().toString()
                + " in the class "
                + TreeUtils.className(trees
                    .getElement(getCurrentPath()))
                + " ["
                + compilationUnitTree.getSourceFile().getName()
                + "] "+" Refer Rules "+ SQLRuleRepository.getRule(Operation.DELETE) + ↩
                     " in the Rule Table");
      }
    }
  }

}
```

Listing A.7: The JdbcCodeAnalyzerVisitor.java class.

```
error: Use Data Store create entity operations on line 25 for the jdbc query string  ↩
    variable preparedStatement in the method addUser in the class com. ↩
    programmingfree.dao.CrudDao [ E:\NCIRL\dissertation\workspace\ ↩
    AjaxCrudjTableSample\src\com\programmingfree\dao\CrudDao.java ]

error: Use Data Store delete entity operations on line 41 for the jdbc query string  ↩
    variable preparedStatement in the method deleteUser in the class com. ↩
    programmingfree.dao.CrudDao [ E:\NCIRL\dissertation\workspace\ ↩
    AjaxCrudjTableSample\src\com\programmingfree\dao\CrudDao.java ]

```

```
error: Use Data Store update entity operations on line 53 for the jdbc query string variable preparedStatement in the method updateUser in the class com.programmingfree.dao.CrudDao [ E:\NCIRL\dissertation\workspace\AjaxCrudjTableSample\src\com\programmingfree\dao\CrudDao.java ]
error: Use Data Store retrieve entity operations on line 71 for the jdbc query string variable rs in the method getAllUsers in the class com.programmingfree.dao.CrudDao [
E:\NCIRL\dissertation\workspace\AjaxCrudjTableSample\src\com\programmingfree\dao\CrudDao.java ]

error: Use Data Store retrieve entity operations on line 90 for the jdbc query string variable preparedStatement in the method getUserById in the class com.programmingfree.dao.CrudDao [ E:\NCIRL\dissertation\workspace\AjaxCrudjTableSample\src\com\programmingfree\dao\CrudDao.java ]
```

Listing A.8: The error logs generated by the SQL Query Analyzer Program for the AjaxCrudJTable application.

A.1.3 Configuration properties file

```
# The configuration properties file for the petclinic application

# The configuration property pointing to the petclinic project's root folder
# used by the GAE BlackList Analyzer Program.
project.rootfolder.location=E:/NCIRL/dissertation/workspace/springpetclinic-maven-appengine

# The configuration property pointing to the petclinic project's Data Access Layer
# used by the SQL Query Analyzer Program.
project.jdbc.files.location=E:/NCIRL/dissertation/workspace/springpetclinic-maven-appengine/src/main/java/org/springframework/samples/petclinic/repository/jdbc
```

Listing A.9: The configuration properties file

A.1.4 Rule table generated by the SQL Query Analyzer Program

The developers can use the rule table displayed by the SQL Query Analyzer Program to map the JDBC APIs to Datastore APIs during the refactoring of SQL Queries.

No.	Op.	JDBC API	Datastore API

1	CREATE	Connection connection = // obtain a connection
		Statement stmt = connection.createStatement();
		stmt.executeUpdate(insert into tablename (column1, column2, column3,...) values (value1, value2, value3,...)
		stmt.close();
			DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
			Entity entity = new Entity (”EntityName”);
			entity.setProperty(”propertyName1”, value1);
			entity.setProperty(”propertyName2”, value2);
			datastore.put(entity);

No.	Op.	JDBC API	Datastore API

2	CREATE	Connection connection = // obtain a connection
		PreparedStatement connection = connection.prepareStatement ("insert into tablename (column1, column2) values (?,?)");
		psmt.setInt(1, value1);
		psmt.setString(2, value2);
		psmt.addBatch ();
		psmt.setInt (1, value1);
		psmt.setString (2, value2);
		pstmt.addBatch ();
		int[] updateCounts = pstmt.executeBatch();
			DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
			Entity entity1 = new Entity("Employee");
			entity1.setProperty("propertyName1", value1);
			entity1.setProperty("propertyName2", value2);
			Entity entity2 = new Entity("Employee");
			Entity2.setProperty("propertyName1", value1);
			entity2.setProperty("propertyName2", value2);
			List¡Entity¿ entityList = Arrays.asList(entity1, entity2);
			datastore.put(entityList);

No.	Op.	JDBC API	Datastore API

3	READ	Connection connection = // obtain a connection
		Statement stmt = connection.createStatement();
		ResultSet rs = stmt.executeQuery(select * from table-name;);
		while (rs.next()) {
		int id = rs.getInt(”columnName1”);
		String id = rs.getString(”columnName2”);
		}
		rs.close();
			DatastoreService datastore = DatastoreServiceFac-tory.getDatastoreService();
			// Use class Query to assemble a query
			Query q = new Query(”EntityName”);
			// Use PreparedQuery interface to retrieve results
			PreparedQuery pq = datastore.prepare(q);
			// Iterate through the entity list
			for (Entity result : pq.asIterable()) {
			String value1 = (String)result.getProperty(”propertyName1”);
			String value2 = (String)result.getProperty(”propertyName2”);
			Long value3 = (Long) result.getProperty(”propertyName3”); }

No.	Op.	JDBC API	Datastore API

4	READ	Connection connection = // obtain a connection
		Statement stmt = connection.createStatement();
		ResultSet rs = stmt.executeQuery(select * from tablename where columnname operator value;);
		while (rs.next())
		int id = rs.getInt(”columnName1”);
		String id = rs.getString(”columnName2”);
		
		rs.close();
			DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
			// Use class Query to assemble a query
			Query q = new Query(”EntityName”);
			Query q = new Query(”EntityName”).addFilter(”propertyName”,Query.FilterOperator,value);
			// Use PreparedQuery interface to retrieve results
			PreparedQuery pq = datastore.prepare(q);

No.	Op.	JDBC API	Datastore API

5	READ	Connection connection = // obtain a connection
		Statement stmt = connection.createStatement();
		ResultSet rs = stmt.executeQuery(select * from table-name where columnname between value1 and value2;)
		while (rs.next()) {
		int id = rs.getInt(”columnName1”);
		String id = rs.getString(”columnName2”); }
		rs.close();
			DatastoreService datastore = DatastoreServiceFac-tory.getDatastoreService();
			Filter rangeFilterCriteria1 = new FilterPredi-cate(”propertyName1”,FilterOperator.GREATERTHANOREQUAL, value1);
			Filter rangeFilterCriteria2 = new FilterPredi-cate(”propertyName2”,FilterOperator.LESSTHANOREQUAL, value2);
			//Use CompositeFilter to combine multiple filters
			Filter rangeFilter = CompositeFilterOperator.and(rangeFilterCriteria1, rangeFilterCriteria2);
			// Use class Query to assemble a query
			Query q = new Query(”EntityName”).setFilter(rangeFilter);
			// Use PreparedQuery interface to retrieve results
			PreparedQuery pq = datastore.prepare(q);
			// Iterate through the entity list
			for (Entity result : pq.asIterable()) {
			String value1 = (String)result.getProperty(”propertyName1”); String value2 = (String)result.getProperty(”propertyName2”); Long value3 = (Long) result.getProperty(”propertyName3”); }

No.	Op.	JDBC API	Datastore API

6	Update	Connection connection = // obtain a connection
		Statement stmt = connection.createStatement();
		stmt.executeUpdate(update tablename set column1=value1,column2=value2,.... where column=value;)
		stmt.close();
			DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
			// load an existing entity from the datastore
			Entity entity = ... (Loaded from the DataStore)
			entity.setProperty(”propertyName1”, newValue1);
			entity.setProperty(”propertyName2”, newValue2);
			// update the entity with the new property values.
			datastore.put(entity);

7	Delete	Connection connection = // obtain a connection
		Statement stmt = connection.createStatement();
		stmt.executeUpdate(delete from tablename where columnname=value;);
			DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
			// load an existing entity from the Datastore
			Entity entity ....
			// update the entity with the new property values.
			datastore.delete(entity.getKey());

A.2 Database Migration Tool important Java classes and configuration files

A.2.1 Database Migration Tool main Java class

```java
package com.springframework.datamigration;

import java.util.Calendar;
import java.util.Map;
import java.util.TimeZone;
import java.util.concurrent.CountDownLatch;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;

import com.springframework.datamigration.exporter.DataExporter;
import com.springframework.datamigration.importer.DataImporter;

/**
 * @author Prasanth M P.
 *
 * The main class for beginning the migration.
 */

public class SpringDataMigration {

  public static ApplicationContext ctx;

  public static Map<String, String> map;

  /**
   * The main method that serves as the entry point for starting the migration
   * process.
   *
   * @param args
   */
  public static void main(String args[]) {

    try {

      loadApplicationContext();
      init();
      System.out.println("************************************************");
      System.out.println("Starting Data Migration");
      System.out.println("");
    Calendar cal1 = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
    long time1 = cal1.getTimeInMillis();

    CountDownLatch latch1 = new CountDownLatch(1);
    DataExporter dataExporterThread = new DataExporter();
    dataExporterThread.setContext(ctx);
    dataExporterThread.setCountDownLatch(latch1);
    Thread t1 = new Thread(dataExporterThread);
    t1.start();
    latch1.await();
    CountDownLatch latch2 = new CountDownLatch(1);

    System.out.println("");
    System.out.println("Exporting Database Tables to CSV files is Completed and now Start to Export CSV files to GAE Entities");
    System.out.println("");

    DataImporter dataImporterThread = new DataImporter();
    dataImporterThread.setContext(ctx);
    dataImporterThread.setCountDownLatch(latch2);
    Thread t2 = new Thread(dataImporterThread);
    t2.start();
    latch2.await();

    Calendar cal2 = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
    long time2 = cal2.getTimeInMillis();
    System.out.println("Time in millisecond " + (time2 - time1));
    System.out.println("");
    System.out.println("Completed Data Migration");
    System.out.println("***********************************************");

   } catch (InterruptedException e) {
    e.printStackTrace();
   } catch (Exception e) {
    e.printStackTrace();
   }
  }

  /**
   * The method creates two tables named DATA_EXPORT_RESULT and
   * DATA_IMPORT_RESULT in the database schema. These DATA_EXPORT_RESULT table
   * will contains information about the Data Export such as the tables
   * exported, total number of records exported, the status of the data export
   * and finally the date of execution.
   *
   * These DATA_IMPORT_RESULT table will contains information about the data
   * import such as the names of the entity kind created in GAE Datastore,
```

```java
   * total number of entites created for each entity kind, the status of the
   * data export and finally the date of execution.
   *
   */
  private static void init() {
    JdbcTemplate jdbcTemplate = ctx.getBean("jdbcTemplate",
        JdbcTemplate.class);

    try {
      jdbcTemplate.execute("DROP TABLE DATA_EXPORT_RESULT");
    } catch (Exception e) {}

    try {
      jdbcTemplate.execute("DROP TABLE DATA_IMPORT_RESULT");
    } catch (Exception e) {}

    jdbcTemplate.execute("CREATE TABLE DATA_EXPORT_RESULT ("
        + "TABLE_NAME VARCHAR(50)," + "ROWS_EXPORTED_COUNT INTEGER,"
        + "ROWS_EXPORT_STATUS VARCHAR(100),"
        + "ROWS_EXPORTATION_DATE DATE)");

    jdbcTemplate.execute("CREATE TABLE DATA_IMPORT_RESULT ("
        + "ENTITY_NAME VARCHAR(50),"
        + "ENTITIES_CREATED_COUNT INTEGER,"
        + "ENTITIES_CREATION_STATUS VARCHAR(100),"
        + "ENTITIES_CREATION_DATE DATE)");
  }

  /**
   * The method loads the application configuration files prior to execution.
   */
  public static void loadApplicationContext() {
    ctx = new ClassPathXmlApplicationContext("SpringDatabaseMigration.xml");
  }
}
```

Listing A.10: SpringDataMigration.java class

A.2.2 Data Exporter Module important Java classes

```java
package com.springframework.datamigration.exporter;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import org.springframework.context.ApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;

/**
 * @author Prasanth M P
 */
public class DataExporter implements Runnable {

  private ApplicationContext context;

  private CountDownLatch countDownLatch;


  /**
   *  The run() method create a fixed size thread pool of worker threads.
   *  The worker threads exports tables to CSV files
   */
  public void run() {
    Properties configurationProperties = (Properties) context.getBean(" ↩
        threadPoolPropertiesConfiguration");
    ExecutorService executorService = Executors.newFixedThreadPool( Integer.valueOf( ↩
        configurationProperties.getProperty("exportThreadPoolSize")));
    List<String> databaseTables = getDatabaseTableNames();
    CountDownLatch exporterCountLatch = new CountDownLatch(
        databaseTables.size());
    Collection<Future<?>> futures = new LinkedList<Future<?>>();
    for (String tableName : databaseTables) {
      TableExporter tableExporterBean = (TableExporter) context.getBean(" ↩
          tableExporter");
      tableExporterBean.setTableName(tableName.toUpperCase());
      tableExporterBean.setCountDownLatch(exporterCountLatch);
      futures.add(executorService.submit(tableExporterBean));
    }
    executorService.shutdown();
    try {
      for (Future<?> future:futures) {
```

```
      future.get(); // cause the current thread to wait for the table export tasks to finish.
    }
    exporterCountLatch.await();
    countDownLatch.countDown();
  } catch (InterruptedException e) {
    e.printStackTrace();
  } catch (ExecutionException e) {
    e.printStackTrace();
  }
}

/**
 * The method returns a list containing the names of all tables in the database.
 * @return List<String>
 */
private List<String> getDatabaseTableNames() {
  final List<String> tablenames = new ArrayList<String>();
  JdbcTemplate jdbcTemplate = (JdbcTemplate) context
      .getBean("jdbcTemplate");
  String showTables = "SHOW TABLES";
  jdbcTemplate.query(showTables, new ResultSetExtractor<List<String>>() {
    public List<String> extractData(ResultSet rs) throws SQLException,
        DataAccessException {
      while (rs.next()) {
        tablenames.add(rs.getString(1));
      }
      return tablenames;
    }
  });

  tablenames.remove("data_export_result");
  tablenames.remove("data_import_result");
  return tablenames;
}

/**
 * The getter method.
 */
public ApplicationContext getContext() {
  return context;
}

/**
 * The setter method.
 */
public void setContext(ApplicationContext context) {
```

```java
    this.context = context;
  }

  /**
   * The getter method.
   */
  public CountDownLatch getCountDownLatch() {
    return countDownLatch;
  }

  /**
   * The setter method.
   */
  public void setCountDownLatch(CountDownLatch countDownLatch) {
    this.countDownLatch = countDownLatch;
  }
}
```

Listing A.11: DataExporter.java class

```java
package com.springframework.datamigration.exporter;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.CountDownLatch;

import org.apache.commons.io.FileUtils;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;

import com.springframework.datamigration.utils.Status;
import com.springframework.datamigration.utils.Utils;

/**
 * @author Prasanth M P
 */
public class TableExporter implements Runnable {

  private CountDownLatch countDownLatch;
```

```

@Value("${fetchSize}")
protected int fetchSize;

private JdbcTemplate jdbcTemplate;

@Value("${migrationfolder}")
protected String migrationFolder;

private int recordCount;

private String tableColumnDatabaseTypes;

private String tableColumnNames;

private String tableName;

/**
 * The run() method contains the workflow logic for exporting the records in
 * the table to CSV files and also for logging the result of export to the
 * database.
 */
public void run() {

  System.out.println("Starting to export Data from Table [ "
      + getTableName() + " ] to CSV files");
  // getRecordCount();
  populateTableRecordCount();
  populateTableMetaData();
  try {
    exportToCSV();
    updateExecutionStatus(tableName, recordCount, Status.SUCCESS,
        new Date());
  } catch (Exception e) {
    updateExecutionStatus(tableName, null, Status.FAILURE, new Date());
  }
  countDownLatch.countDown();
  System.out.println("Finished exporting Data from Table [ "
      + getTableName() + " ] to CSV files");
}

/**
 * The method gets the count of total number of records in the table.
 */
private void populateTableRecordCount() {
  int recordCount = getJdbcTemplate().queryForInt(getRecordCountQuery());
  setRecordCount(recordCount);
```

```
}

/**
 * The method generates the meta data for inserting in each CSV export file.
 * The meta data involves the colum names and column types.
 */
private void populateTableMetaData() {
  String jdbcTableMetaDataQuery = getTableMetaDataQuery();
  final List<String> columnName = new ArrayList<String>();
  final List<String> columnType = new ArrayList<String>();
  getJdbcTemplate().query(jdbcTableMetaDataQuery,
      new RowMapper<String>() {
        public String mapRow(ResultSet rs, int rowNum)
            throws SQLException {
          columnName.add(rs.getString(1));
          columnType.add(rs.getString(2));
          return null;
        }
      });

  setTableColumnNames(Utils.getCSV(columnName));
  setTableColumnDatabaseTypes(Utils.getCSV(columnType));
}


/**
 * The method creates CSV files and export the records in the table in
 * batches.
 */
private void exportToCSV() {
  int noCSVFiles = csvFilesPerTable();
  prepareDirectory();
  int lowerLimit = 0;
  PrintWriter pw = null;
  for (int i = 0; i < noCSVFiles; i++) {
    File file = new File(this.migrationFolder + "\\" + getFolderName(),
        getFileNamePrefix() + i + ".csv");
    try {
      file.createNewFile();
      pw = new PrintWriter(file);
    } catch (IOException e2) {
      e2.printStackTrace();
    }
    String fileContentsToWrite = getFileContentToWrite(lowerLimit);
    pw.write(fileContentsToWrite);
    pw.flush();
    pw.close();
```

```
      lowerLimit = lowerLimit + fetchSize;
    }
  }

  /**
   * Indicates how many CSV files will be created for storing the records in
   * the table. The number of CSV files depends on the fetch size configure.
   * More the fetch size lesser the number of CSV files.
   *
   * @return integer
   */
  public int csvFilesPerTable() {
    int noCSVFiles = 0;
    if (getRecordCount() < getFetchSize()) {
      noCSVFiles = 1;
    } else if (getRecordCount() % getFetchSize() == 0) {
      noCSVFiles = getRecordCount() / getFetchSize();
    } else {
      noCSVFiles = getRecordCount() / getFetchSize() + 1;
    }
    return noCSVFiles;
  }

  /**
   * The method just update the status of the export of the table.
   *
   * @param tableName - The name of the table in the database exported as CSV file
   * @param recordCount - The number of records that are exported.
   * @param status - The status of the export of the table
   * @param date - The date on which the table exported.
   */
  private void updateExecutionStatus(String tableName, Integer recordCount,
      Status status, Date date) {
    final String INSERT_SQL = "INSERT INTO DATA_EXPORT_RESULT (TABLE_NAME,"
        + "ROWS_EXPORTED_COUNT," + "ROWS_EXPORT_STATUS,"
        + "ROWS_EXPORTATION_DATE) VALUES (?,?,?,?)";
    jdbcTemplate.update(INSERT_SQL, tableName, recordCount, status.name(),
        new java.sql.Date(date.getTime()));
  }

  /**
   * The method fetches a batch of records and parse the records in CSV format
   * to be returned.
   *
   * @param lowerLimit - the lower limit used to calculate the range of records to ↩
       be
   *           fetched for exporting. range equal to (lowerLimit -->
```

```java
 *          lowerLimit+fetchSize)
 * @return String - returns the fetched records in CSV format to be written
 *       to a CSV file.
 */
public String getFileContentToWrite(int lowerLimit) {
  return getJdbcTemplate().query(getQuery(),
      new Object[] { lowerLimit, fetchSize },
      new ResultSetExtractor<String>() {
        public String extractData(ResultSet rs)
            throws SQLException, DataAccessException {
          StringBuffer fileContentsToWrite = null;
          fileContentsToWrite = new StringBuffer();
          fileContentsToWrite.append(getTableColumnNames());
          fileContentsToWrite.append("\n");
          fileContentsToWrite
              .append(getTableColumnDatabaseTypes());
          fileContentsToWrite.append("\n");
          List<String> row = null;
          while (rs.next()) {
            row = new ArrayList<String>();
            for (int columnNo = 1; columnNo <= rs.getMetaData()
                .getColumnCount(); columnNo++) {
              row.add(rs.getString(columnNo));
            }
            fileContentsToWrite.append(Utils.getCSV(row)
                .concat("\n"));
          }
          return fileContentsToWrite.toString();
        }
      });
}

/**
 * Creates a directory for each table to be exported where the related CSV
 * files will be placed.
 */
public void prepareDirectory() {
  File dir = new File(this.migrationFolder + "\\" + getFolderName());
  if (dir.exists()) {
    try {
      FileUtils.deleteDirectory(dir);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  dir.mkdir();
}
```

```

  //getter method
  public CountDownLatch getCountDownLatch() {
    return countDownLatch;
  }

  //setter method
  public void setCountDownLatch(CountDownLatch countDownLatch) {
    this.countDownLatch = countDownLatch;
  }

  //getter method
  public int getFetchSize() {
    return fetchSize;
  }

  //setter method
  public void setFetchSize(int fetchSize) {
    this.fetchSize = fetchSize;
  }

  //getter method
  public String getFileNamePrefix() {
    return tableName.toUpperCase();
  }

  //getter method
  public String getFolderName() {
    return tableName.toUpperCase();
  }

  //getter method
  public JdbcTemplate getJdbcTemplate() {
    return jdbcTemplate;
  }

  //setter method
  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  //getter method
  public String getMigrationFolder() {
    return migrationFolder;
  }

  //setter method
  public void setMigrationFolder(String migrationFolder) {
    this.migrationFolder = migrationFolder;
  }

  //getter method
  public String getQuery() {
    return Utils.getTableRecordSelectQuery(tableName);
  }

  //getter method
  public int getRecordCount() {
    return recordCount;
  }

  //setter method
  public void setRecordCount(int recordCount) {
    this.recordCount = recordCount;
  }

  //getter method
  public String getRecordCountQuery() {
    return Utils.getTableRecordCountQuery(this.tableName);
  }

  //getter method
  public String getTableColumnDatabaseTypes() {
    return tableColumnDatabaseTypes;
  }

  //setter method
  public void setTableColumnDatabaseTypes(String tableColumnDatabaseTypes) {
    this.tableColumnDatabaseTypes = tableColumnDatabaseTypes;
  }

  //getter method
  public String getTableColumnNames() {
    return tableColumnNames;
  }

  //setter method
  public void setTableColumnNames(String tableColumnNames) {
    this.tableColumnNames = tableColumnNames;
  }

  //getter method
  public String getTableMetaDataQuery() {
    return Utils.getTableMetaDataQuery(this.tableName);
```

```java
  }

  //getter method
  public String getTableName() {
    return tableName;
  }

  //setter method
  public void setTableName(String tableName) {
    this.tableName = tableName;
  }
}
```

Listing A.12: TableExporter.java class

A.2.3 Data Importer Module important Java classes

```java
package com.springframework.datamigration.exporter;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

import org.springframework.context.ApplicationContext;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;

/**
 * @author Prasanth M P
 */
public class DataExporter implements Runnable {

  private ApplicationContext context;

  private CountDownLatch countDownLatch;


  /**
   *  The run() method create a fixed size thread pool of worker threads.
   *  The worker threads exports tables to CSV files
   */
  public void run() {
    Properties configurationProperties = (Properties) context.getBean("threadPoolPropertiesConfiguration");
    ExecutorService executorService = Executors.newFixedThreadPool( Integer.valueOf( configurationProperties.getProperty("exportThreadPoolSize")));
    List<String> databaseTables = getDatabaseTableNames();
    CountDownLatch exporterCountLatch = new CountDownLatch(
        databaseTables.size());
    Collection<Future<?>> futures = new LinkedList<Future<?>>();
    for (String tableName : databaseTables) {
      TableExporter tableExporterBean = (TableExporter) context.getBean("tableExporter");
      tableExporterBean.setTableName(tableName.toUpperCase());
      tableExporterBean.setCountDownLatch(exporterCountLatch);
      futures.add(executorService.submit(tableExporterBean));
    }
    executorService.shutdown();
    try {
      for (Future<?> future:futures) {
        future.get(); // cause the current thread to wait for the table export tasks to finish.
      }
      exporterCountLatch.await();
      countDownLatch.countDown();
    } catch (InterruptedException e) {
      e.printStackTrace();
    } catch (ExecutionException e) {
      e.printStackTrace();
    }
  }

  /**
   * The method returns a list containing the names of all tables in the database.
   * @return List<String>
   */
  private List<String> getDatabaseTableNames() {
    final List<String> tablenames = new ArrayList<String>();
    JdbcTemplate jdbcTemplate = (JdbcTemplate) context
        .getBean("jdbcTemplate");
    String showTables = "SHOW TABLES";
    jdbcTemplate.query(showTables, new ResultSetExtractor<List<String>>() {
      public List<String> extractData(ResultSet rs) throws SQLException,
          DataAccessException {
        while (rs.next()) {
          tablenames.add(rs.getString(1));
        }
        return tablenames;
      }
    });

    tablenames.remove("data_export_result");
    tablenames.remove("data_import_result");
    return tablenames;
  }

  /**
   * The getter method.
   */
  public ApplicationContext getContext() {
    return context;
  }

  /**
   * The setter method.
   */
  public void setContext(ApplicationContext context) {
    this.context = context;
  }

  /**
   * The getter method.
   */
  public CountDownLatch getCountDownLatch() {
    return countDownLatch;
  }

  /**
   * The setter method.
   */
  public void setCountDownLatch(CountDownLatch countDownLatch) {
    this.countDownLatch = countDownLatch;
  }
}
```

Listing A.13: DataImporter.java class

```
package com.springframework.datamigration.exporter;

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.CountDownLatch;

import org.apache.commons.io.FileUtils;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.core.RowMapper;

import com.springframework.datamigration.utils.Status;
import com.springframework.datamigration.utils.Utils;

/**
 * @author Prasanth M P
 */
public class TableExporter implements Runnable {

  private CountDownLatch countDownLatch;

  @Value("${fetchSize}")
  protected int fetchSize;

  private JdbcTemplate jdbcTemplate;

  @Value("${migrationfolder}")
  protected String migrationFolder;

  private int recordCount;

  private String tableColumnDatabaseTypes;

  private String tableColumnNames;

  private String tableName;

  /**
   * The run() method contains the workflow logic for exporting the records in
   * the table to CSV files and also for logging the result of export to the
   * database.
   */
  public void run() {

    System.out.println("Starting to export Data from Table [ "
        + getTableName() + " ] to CSV files");
    // getRecordCount();
    populateTableRecordCount();
    populateTableMetaData();
    try {
      exportToCSV();
      updateExecutionStatus(tableName, recordCount, Status.SUCCESS,
          new Date());
    } catch (Exception e) {
      updateExecutionStatus(tableName, null, Status.FAILURE, new Date());
    }
    countDownLatch.countDown();
    System.out.println("Finished exporting Data from Table [ "
        + getTableName() + " ] to CSV files");
  }

  /**
   * The method gets the count of total number of records in the table.
   */
  private void populateTableRecordCount() {
    int recordCount = getJdbcTemplate().queryForInt(getRecordCountQuery());
    setRecordCount(recordCount);
  }

  /**
   * The method generates the meta data for inserting in each CSV export file.
   * The meta data involves the colum names and column types.
   */
  private void populateTableMetaData() {
    String jdbcTableMetaDataQuery = getTableMetaDataQuery();
    final List<String> columnName = new ArrayList<String>();
    final List<String> columnType = new ArrayList<String>();
    getJdbcTemplate().query(jdbcTableMetaDataQuery,
        new RowMapper<String>() {
          public String mapRow(ResultSet rs, int rowNum)
              throws SQLException {
            columnName.add(rs.getString(1));
            columnType.add(rs.getString(2));
            return null;
          }
        });

    setTableColumnNames(Utils.getCSV(columnName));
    setTableColumnDatabaseTypes(Utils.getCSV(columnType));
  }
```

```java


/**
 * The method creates CSV files and export the records in the table in
 * batches.
 */
private void exportToCSV() {
  int noCSVFiles = csvFilesPerTable();
  prepareDirectory();
  int lowerLimit = 0;
  PrintWriter pw = null;
  for (int i = 0; i < noCSVFiles; i++) {
    File file = new File(this.migrationFolder + "\\" + getFolderName(),
        getFileNamePrefix() + i + ".csv");
    try {
      file.createNewFile();
      pw = new PrintWriter(file);
    } catch (IOException e2) {
      e2.printStackTrace();
    }
    String fileContentsToWrite = getFileContentToWrite(lowerLimit);
    pw.write(fileContentsToWrite);
    pw.flush();
    pw.close();
    lowerLimit = lowerLimit + fetchSize;
  }
}

/**
 * Indicates how many CSV files will be created for storing the records in
 * the table. The number of CSV files depends on the fetch size configure.
 * More the fetch size lesser the number of CSV files.
 *
 * @return integer
 */
public int csvFilesPerTable() {
  int noCSVFiles = 0;
  if (getRecordCount() < getFetchSize()) {
    noCSVFiles = 1;
  } else if (getRecordCount() % getFetchSize() == 0) {
    noCSVFiles = getRecordCount() / getFetchSize();
  } else {
    noCSVFiles = getRecordCount() / getFetchSize() + 1;
  }
  return noCSVFiles;
}

```

```java
/**
 * The method just update the status of the export of the table.
 *
 * @param tableName - The name of the table in the database exported as CSV file
 * @param recordCount - The number of records that are exported.
 * @param status - The status of the export of the table
 * @param date - The date on which the table exported.
 */
private void updateExecutionStatus(String tableName, Integer recordCount,
    Status status, Date date) {
  final String INSERT_SQL = "INSERT INTO DATA_EXPORT_RESULT (TABLE_NAME,"
      + "ROWS_EXPORTED_COUNT," + "ROWS_EXPORT_STATUS,"
      + "ROWS_EXPORTATION_DATE) VALUES (?,?,?,?)";
  jdbcTemplate.update(INSERT_SQL, tableName, recordCount, status.name(),
      new java.sql.Date(date.getTime()));
}

/**
 * The method fetches a batch of records and parse the records in CSV format
 * to be returned.
 *
 * @param lowerLimit - the lower limit used to calculate the range of records to ↩
    be
 *           fetched for exporting. range equal to (lowerLimit -->
 *           lowerLimit+fetchSize)
 * @return String - returns the fetched records in CSV format to be written
 *        to a CSV file.
 */
public String getFileContentToWrite(int lowerLimit) {
  return getJdbcTemplate().query(getQuery(),
      new Object[] { lowerLimit, fetchSize },
      new ResultSetExtractor<String>() {
        public String extractData(ResultSet rs)
            throws SQLException, DataAccessException {
          StringBuffer fileContentsToWrite = null;
          fileContentsToWrite = new StringBuffer();
          fileContentsToWrite.append(getTableColumnNames());
          fileContentsToWrite.append("\n");
          fileContentsToWrite
              .append(getTableColumnDatabaseTypes());
          fileContentsToWrite.append("\n");
          List<String> row = null;
          while (rs.next()) {
            row = new ArrayList<String>();
            for (int columnNo = 1; columnNo <= rs.getMetaData()
                .getColumnCount(); columnNo++) {
              row.add(rs.getString(columnNo));
          }
          fileContentsToWrite.append(Utils.getCSV(row)
              .concat("\n"));
        }
        return fileContentsToWrite.toString();
      }
    });
}

/**
 * Creates a directory for each table to be exported where the related CSV
 * files will be placed.
 */
public void prepareDirectory() {
  File dir = new File(this.migrationFolder + "\\" + getFolderName());
  if (dir.exists()) {
    try {
      FileUtils.deleteDirectory(dir);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
  dir.mkdir();
}

//getter method
public CountDownLatch getCountDownLatch() {
  return countDownLatch;
}

//setter method
public void setCountDownLatch(CountDownLatch countDownLatch) {
  this.countDownLatch = countDownLatch;
}

//getter method
public int getFetchSize() {
  return fetchSize;
}

//setter method
public void setFetchSize(int fetchSize) {
  this.fetchSize = fetchSize;
}

//getter method
public String getFileNamePrefix() {
  return tableName.toUpperCase();
}

//getter method
public String getFolderName() {
  return tableName.toUpperCase();
}

//getter method
public JdbcTemplate getJdbcTemplate() {
  return jdbcTemplate;
}

//setter method
public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
  this.jdbcTemplate = jdbcTemplate;
}

//getter method
public String getMigrationFolder() {
  return migrationFolder;
}

//setter method
public void setMigrationFolder(String migrationFolder) {
  this.migrationFolder = migrationFolder;
}

//getter method
public String getQuery() {
  return Utils.getTableRecordSelectQuery(tableName);
}

//getter method
public int getRecordCount() {
  return recordCount;
}

//setter method
public void setRecordCount(int recordCount) {
  this.recordCount = recordCount;
}

//getter method
public String getRecordCountQuery() {
  return Utils.getTableRecordCountQuery(this.tableName);
}
```

```java

  //getter method
  public String getTableColumnDatabaseTypes() {
    return tableColumnDatabaseTypes;
  }

  //setter method
  public void setTableColumnDatabaseTypes(String tableColumnDatabaseTypes) {
    this.tableColumnDatabaseTypes = tableColumnDatabaseTypes;
  }

  //getter method
  public String getTableColumnNames() {
    return tableColumnNames;
  }

  //setter method
  public void setTableColumnNames(String tableColumnNames) {
    this.tableColumnNames = tableColumnNames;
  }

  //getter method
  public String getTableMetaDataQuery() {
    return Utils.getTableMetaDataQuery(this.tableName);
  }

  //getter method
  public String getTableName() {
    return tableName;
  }

  //setter method
  public void setTableName(String tableName) {
    this.tableName = tableName;
  }
}
```

Listing A.14: TableImporter.java class

A.2.4 Configuration properties file

```
# The configuration file used by the Database Migration Tool for migrating the table ↩
     records
# from the Database used by the application to GAE Entities in the Datastore.

# MySQL settings
```

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/petclinic
jdbc.username=root
jdbc.password=

# base location at which tables data will be exported as CSV files.
migrationfolder=E:\\NCIRL\\dissertation\\workspace\\MigrationFolder

# properties to configure the thread pool size for exporter and importer.
exportThreadPoolSize=5
importThreadPoolSize=5

# Fetchsize indicates the rate at which the table records will be read from the ↩
    database for export.
fetchSize=500

# Configuration Data for accessing the application's Datastore on the Cloud using ↩
    remote API's.
hostname=spring-petclinic.appspot.com
port=443
userEmail=prasanthmp500@gmail.com
password=**********
```

Listing A.15: The configuration properties file used by the Database Migration Tool

A.2.5 The table name to entity kind mapping XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www. ↩
           springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context http://www.springframework.org/ ↩
          schema/context/spring-context.xsd">

<!-- The sample mapping file used to map the tables in the RDBMS to Entity Kinds in ↩
    the GAE Datastore -->
<bean id="tableToEntityMapping" class="org.springframework.beans.factory.config. ↩
    MapFactoryBean">
  <property name="sourceMap">
    <map>
    <entry key="VISITS" value="Visit"/>
    <entry key="VETS" value="Vet"/>
    <entry key="VET_SPECIALTIES" value="VetSpecialties"/>
    <entry key="TYPES" value="Types"/>
    <entry key="SPECIALTIES" value="Specialties"/>
    <entry key="PETS" value="Pet"/>
    <entry key="OWNERS" value="Owner"/>
    </map>
  </property>
</bean>

</beans>
```

Listing A.16: The mapping.xml file used for mapping the table name to an entity kind in the GAE. Datastore