Dribble: A Geosocial Networking System – Server-side

Author: Daniel Mankowitz – 0616159H

Other Members:

A. Paverd – 0702663D

C. Epstein – 0706965X

A. Campbell - 0605245J

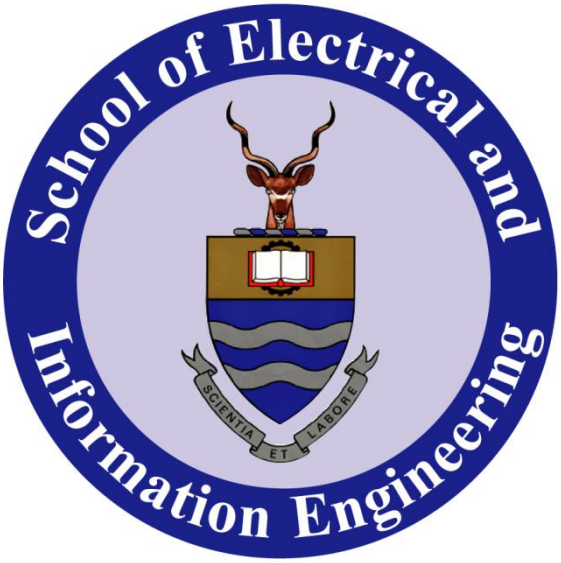
G. Favish - 0713568E

Course Name: Software Development III

Course Code: ELEN4010

Date: 03 May 2010

http://github.com/ajpaverd/Dribble



School of Electrical and Information Engineering

University of the Witwatersrand

DRIBBLE: A GEOSOCIAL NETWORKING SYSTEM - SERVER SIDE

D. Mankowitz

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract: A server-side Dribble application has been implemented using the JEE framework and has been deployed on a Glassfish application server. A standard communication protocol has been developed to transmit messages, or Dribs, and topics, or DribSubjects, to and from the client application. The communication is purely object-oriented which allows for higher level communication between both the client and the server application. RESTful web services have been implemented for communication as well as message-driven enterprise beans to process the data. A unique ID algorithm and a dynamically changing popularity score have been created for Dribs and DribSubjects. The testing was conducted in netbeans and using the Glassfish logger. The server-side application successfully integrated with the mobile device.

Key words: Drib, Dribble, Enterprise JavaBean, Java Enterprise Edition, Location-based,

1. INTRODUCTION

The focus of this project was to build and release an open-source software application. An application called Dribble has been developed that conforms to the imposed project specifications and is based on a social networking structure. This application allows users to upload and download messages to and from their mobile phone that are within the range of a user's current location. This provides relevant and informative feedback to user's of the application. The application consists of a client application and a server-side Java Enterprise Edition (JEE) application residing on an application server. The server-side application is discussed in detail in this report.

Initially a background on server-side development using the JEE framework is discussed in *Section 2.* This is followed by the requirements and success criteria for the server-side application in *Section 3.* Various assumptions were made and constraints imposed upon the application design and these are detailed in *Section 4.* An overview of the server-side system is presented in *Section 5.* This is followed by an in-depth analysis of the Application server in *Section 7.* The web services architecture is then detailed *Section 8.* This includes an in depth description of RESTful web services and their implementation in the application. Enterprise JavaBeans (EJB) and processing implemented in the application are detailed in *Section 9.* The project underwent a testing phase and the results of this phase are reviewed in *Section 10.* The time management aspect of the project is detailed in *Section 11.* A critical analysis of the project has been conducted in *Section 12.* which includes discussion of the trade-offs as well as recommendations for improvements on similar projects of the same nature. The report is finally concluded in *Section 13.*

2. BACKGROUND

The server-side application has been developed using the Java Enterprise Edition (JEE) application framework. This framework has a multi-tiered distributed application model [1]. Thus the development of an application under this framework involves a number of tiers with separate functionality. JEE applications are made up of components. A component is a self-contained functional software unit [1]. This unit is assembled with other components into a JEE application. A big advantage in using this type of framework is that much of the code is organised into reusable units which aids to ease of development.

Enterprise JavaBeans implement the business logic in a JEE application. There are two types of beans, session beans and message-driven beans [1]. This application implements message-driven beans.

3. REQUIREMENTS AND SUCCESS CRITERIA

The requirements determined by the group developing the server-side application are as follows. The application needs to be implemented using the JEE framework. The application needs to be completed within the allotted time-frame. The application requires some form of information storage such as a database, processing and calculations. Enterprise JavaBeans need to be investigated and implemented to perform the business logic. The application needs to be deployed on an application server. The application needs to be a modular design that clearly indicates a multi-tier architecture. Based on the following requirements, the success criteria is to create a multi-tiered server-side application that is implemented using the JEE framework on an application server such as Glassfish. The application must be completed within the allotted time and must integrate effectively with the mobile device.

4. ASSUMPTION AND CONSTRAINTS

An assumption made is that there is always an available connection to the server to which the clients have access. The server application needs to be scalable as there is the potential for a large user base. The server needs to be able to perform its own load balancing. Another assumption is that the server will manage the applications as is necessary and will not fail. Thus another assumption that the server will be fault tolerant has been made. The database is also assumed to have sufficient storage to maintain a large user base.

There are also some constraints imposed upon the system. The connection between the client and server is limited by the bandwidth of the connection. There is a limited amount of connections that can be made to the server simultaneously. The application is constrained to the JEE framework. The application is also limited by the processing speed of the server that it is residing on.

5. SERVER-SIDE OVERVIEW

The application itself resides on an application server and has a structure as seen in *Figure 1*. The application consists of a web tier where the web services are implemented. These are the 'putDribResource', 'getDribsResource' and 'getDribSubjectsResource' web services. The web services connect to the EJBs using queues and synchronous queues that will be discussed in depth in sections 8.2 and 8.3 respectively. The application contains three EJBs namely 'putDribBean', 'getDribsBean' and 'getDribSubjectsBean'. These beans handle all the processing of data and use helper classes to perform more complex processing algorithms. Finally a composition relationship exists between the database and the beans where data is sent to and retrieved from the database using instances of the dataset and bean objects.

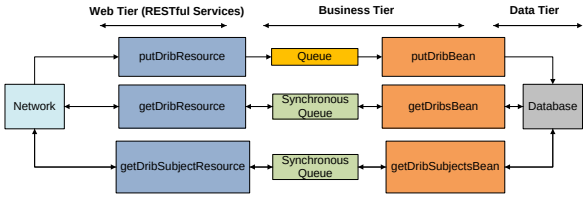


Figure 1 : The server-side flow diagram

5.1 Modularity

One of the important features about the design of the application is the design modularity. As can be seen in *Figure 1*, the server application has a web-tier, a business tier and a data tier. This was created to allow components of the application to be independent of one another. The web-tier and enterprise beans on the business-tier are completely decoupled as they are both sending and receiving data using queues as can be seen in *Figure 1*. The EJBs have a composition relationship with the dataset objects and thus a small dependency exists. But since the dataset object is an interface, the EJBs and dataset objects are still largely independent of one another.

6. COMMUNICATION PROTOCOL

The application consists of two fundamental classes, namely a DribSubject and a Drib. These are stored on the 'dribble.common' package in the Dribble_Common Java class library. A DribSubject is a topic and associated with this topic is a number of messages called Dribs. These two objects form the data that is passed between the client and server application. Thus a standard communication protocol had to be developed based on the structure of these objects. Objects can be represented in an XML form using

RESTful web services as will be seen in section 8.

A DribSubject has a name, latitude and longitude coordinates, a subject ID, number of views, number of posts to the subject, time and a popularity score. A Drib consists of some text, latitude and longitude coordinates, a message ID, a DribSubject object, a like count and popularity score. It was decided that a Drib is associated with a DribSubject and hence should contain the DribSubject object. Both classes implement the Serializable interface which allows the objects to be serialised. This will be covered in more depth in section 8.2.

7. APPLICATION SERVER

The Glassfish application server was responsible for store and running the JEE application. The server was administered using the administration console and a number of settings had to implemented to effectively use the server. The Glassfish database was used as the primary data storage for the application. In order to create a connection to the database, a Java Database Connectivity (JDBC) connection pool had to be setup. This is a group of reusable connections for a particular database. The connection pool allowed for a maximum of 32 connections to be established to service clients. In addition to this, a JDBC resource had to be setup. A JDBC resource provides applications with a means of connecting to a database [2]. The JDBC resource is retrieved by using a JNDI lookup corresponding to the resource name. Once retrieved, a JDBC resource is able to specify the connection pool that provides connections to the database. Once the JDBC resource has specified a connection pool for the database, a physical connection with the database can be established [2].

Another important configuration is that of Java Message Service (JMS) resources. Queues, discussed in section 8.2 need to be setup such that the web service resources detailed in section 8.4 can communicate with the business logic within the application. Thus connection factories and queues need to be configured to enable this functionality. This can be achieved using the administrator's console. A screen-shot of the console can be seen in Appendix A, *Figure A.3*

It is possible to manually deploy and undeploy applications in the administration console. Applications can also be enabled, disabled and restarted using the console. Logging can also be configured, log handlers specified and log files created.

8. WEB SERVICES

The server-side application needed an efficient, light-weight method of communicating with the mobile application. These criteria were attained by implementing Representation State Transfer (RESTful) web services. All data and functionality in RESTful web services are considered resources and the resources are accessed using Uniform Resource Identifiers (URIs)[3]. The URIs are generally links that a user will access using HTTP methods such as GET, PUT, POST and DELETE in order to expose a specific RESTful web service. The web services are designed to use a stateless communication protocol which in this case is HTTP [3].

Typical JavaEE modules contain deployment descriptors which are XML documents that describes the deployment settings of an application, module or component [1]. The deployment descriptor is read at run-time by the JEE server and acts upon the application, module or component accordingly.
The JSR 311:JAX-RS API is the API for RESTful web services [1]. It implements Java programming annotations which allow the user to configure a web service with far more ease and efficiency. An annotation is a modifier that injects additional data into a Java class and or method [4]. The annotation removes the need to create an additional deployment descriptor to tell the application or module to do something. The request can be placed directly into the code using the annotation.

The basic RESTful web service contains a root resource class. This class contains all the data and functions that are made available by the service when it is exposed. The RESTful web services were created using 'Simple Root Resource' design patterns. This pattern creates a RESTful root resource class that contains GET and PUT methods. These methods are made available by the JSR-311 API.

Just above the class definition, an annotation @*Path* is found. This identifies the partial URI path template to which the resource responds. The full URI path is that of the server on which the web service is deployed. Thus by accessing the server URI and adding the partial URI path defined by @*Path*, the web service will be exposed.

The @*Context* annotation may also be used in web services. It instantiates an interface UriInfo that provides access to application and request information from URIs [5].

The PUT and GET methods are specified in the resource class by HTTP-specific annotations @*PUT* and @*GET* respectively. These annotations link HTTP methods to Java programming language methods [1]. Thus a method that is preceded by an @*PUT* will be called upon when a client sends a PUT HTTP request to this specific web service. Similarly @*GET* responds to HTTP GET requests for a particular web service.

Two more important annotations are @*Produces* and @*Consumes*. The @*Produces* annotation determines the MIME media type that the web service resource will produce and send back to the client[1]. Similary the @*Consumes* determines the MIME media type that the web service resource is defined to accept.

As can be seen in *Figure 1*, there are three RESTful web services that were implemented, each providing resources that can be accessed by a link to the relevant web service. The 'putDribResource' web service resource receives a Drib object discussed in and adds it to the database. The 'getDribsResource' sends a list of Dribs associated with a specific topic to the client. Finally 'getDribSubjectsResource' sends a list of popular subjects to the client. These web resources are a part of the 'dribble.communications' package and form the web component of the application.

8.1 Object-oriented Communication

As mentioned previously, Dribs and DribSubjects need to be sent to clients and received from clients. Sending and receiving objects using RESTful web services is a very powerful feature that can achieve this object-oriented communication.

The RESTful web services implemented in the Dribble application produce and consume XML. Thus the Drib and DribSubject need to be transformed into XML to be sent via the web service. This is achieved using the @*XmlRootElement* annotation preceding Drib and DribSubject class definitions respectively. This annotation turns the class name into the root element of an XML document and the class variables into XML tags within the root element. This generated XML document, which is a representational form of the Java object, can then be sent to the client. The XML structures of the Drib and DribSubjects respectively can be seen in Appendix A, *Figure A.1* and *Figure A.2* respectively.

There was a problem in sending the XML structures to the client application as the name of the root elements had to conform to the root elements that the clients expected. Thus in order to create flexible naming, XML wrapper classes DribList and DribSubjectList were created. These classes had XML wrappers and contained an ArrayList data member. They used the @*XMLRootElement* annotation to rename the XML root tag to suit the clients needs. Thus these classes effectively created a new hierarchy of root elements as seen in Appendix A, *Figure A.1*. The 'dribble.common.DribList' is the new root element name from the wrapper class DribList. The 'list' results from the ArrayList data member and 'Dribble.common.Drib' represents the XML root element of the original Drib.

8.2 Queueing

Once a web service receives data, it may need to send the data to other objects such as enterprise beans on the business tier for processing. The communication between the web services and enterprise beans as seen in *Figure 1* is achieved using a point-to-point queueing implementation [6]. This type of queueing is unidirectional. Queues are part of the Java Messaging Service (JMS) which is an API that is part of the JEE plaform. This API allows an enterprise system's messages to be created, sent, received and read [7].

The queueing process can be seen in *Figure 2*.

The web service object or client one will send a message to a queue. The queue will then retain the message until the enterprise bean, client two, discussed in 9. consumes the message in order to process it[6]. The object that sends the message into the queue is known as the message producer and object receiving the message is the message consumer[6].

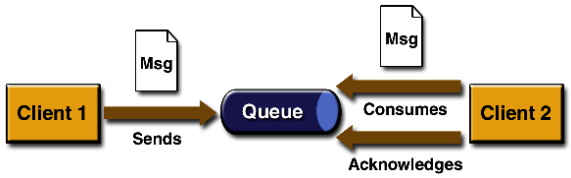


Figure 2 : A JMS queue flow diagram [6]

In order to use a queue, the queue name needs to be looked up using the 'lookup' function of the Java Naming and Directory Interface (JNDI). The JNDI provides naming and directory functionality. This functionality allows a JavaEE application to store and retrieve any type of named Java object [1]. This is because a naming service associates names with objects. A binding is a specific association between a name and an object and the context is a set of these bindings [8]. By binding the name of the queue to the actual queue object within the specific context defined in the application, the web resource can connect to the queue. The context defined in the application's web resource is an InitialContext which implements context and serves as an entry point to the naming system [8].
Once the queue object associated with the lookup name has been retrieved, the QueueConnectionFactory from which the queue may obtain a connection is also retrieved. Subsequently a QueueSession and a QueueSender are created and the queue sender attaches itself to the queue object. The object may now be sent to the queue. The object is serialised into an ObjectMessage and is sent to the queue. An enterprise bean listening to the queue, consumes the serialised object, deserialises the object and processes it.

8.3 Synchronous Queueing

In the case that a message is sent onto a queue and a reply needs to be received, the traditional unidirectional queueing described above cannot be utilised. In this case the concept of synchronous queueing must be used. A queue connection must still be established, but in addition to this a temporary queue destination must be created in the web service resource as seen in *Figure 3.* A QueueReceiver object in the web resource is created and is bound to this temporary destination waiting for a reply. The message is then sent onto the normal queue by the web service and is received by the EJB. The EJB has built in functionality described in section 9.1 to retrieve the message and reply to the web service resource via the temporary destination. In this way bidirectional communication can be achieved.

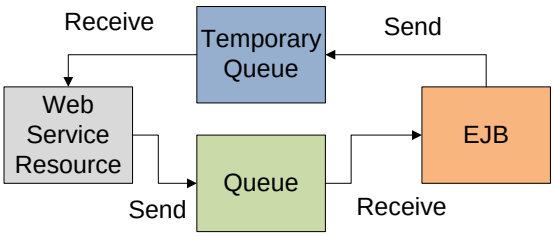


Figure 3 : Synchronous Queueing diagram

8.4 RESTful Web Services Implementation

The 'putDribResource' web service is accessed using an HTTP PUT request. The class itself consists of methods 'putXML' and 'getXML'. The method 'getXML' is not utilised and thus throws an *UnsupprotedOperationException*. 'putXML' is preceded by *@Put* and *@Consumes* annotations. Every time this resource is exposed, a new instance is created, and it constructs a connection to the 'putDribQueue' found on the Glassfish server from within it's constructor. 'putXML' then receives a Drib object argument that has been sent by the client. The object is then serialised and placed on the JMS queue for processing.

'GetDribsResource' creates a queue connection from within it's constructor when a new instance is created. This resource services GET requests, using the 'getXML' method, which contain query parameters of latitude, longitude, results and subjectID. These parameters are used to request a certain amount of Drib messages based on the results value, that are within a specific latitude and longitude for a specific subject ID. The parameters are initially stored in a MultivaluedMap which stores name-key pairs. These query parameters are then passed to the enterprise bean as

object properties via the 'getDribsQueue'. This implementation requires bidirectional communication with the enterprise bean. Thus a synchronous queue has been setup to receive a reply as discussed in section 8.3. The list of Dribs for a particular subject are received by the web resource as an ArrayList. The ArrayList is wrapped in the DribList wrapper class and is sent to the client.

'GetDribSubjectsResource' works in a similar way to 'GetDribsResource' in that a queue connection is created and a temporary queue is created to create bidirectional communication with the business layer. The query parameters sent in the GET request for this resource are latitude, longitude and results. These parameters are used to retrieve a list of subjects depending on the number of results required, from a specific latitude and longitude location.

9. ENTERPRISE BEANS AND PROCESSING

9.1 Message-driven Beans

The business layer of the application consists of the EJB components and helper classes. As can be seen in *Figure 1* there are three EJB components namely putDribBean, getDribsBean and getDribSubjectsBean. These three beans are message-driven beans and they provide the functionality to receive messages asynchronously from JMS queues [1]. These beans are similar to event listeners and can receive messages sent by any JEE component. Message-driven beans maintain no conversational state and instances of the beans can be pooled to allow streams of messages to be processed concurrently making them very scalable[1].

A typical message-driven bean class consists of a number of important annotations. The *@MessageDriven* annotation is used to specify the JMS queue that the bean must listen on [9]. This annotation includes configuration information such as the acknowledgement mode and the type of destination that the bean is listening on. The bean implements the MessageListener interface that contains the 'onMessage' method. This method is invoked when a queue that the bean is listening on receives a message. The message object is received into this method and can be processed.

9.1.1 Bean Implementation The putDribBean listens on the 'putDribQueue' JMS message queue and when a client sends a PUT request, the web resource sends the Drib object to the business layer to be processed. The bean's 'onMessage' method retrieves the object and deserialises the object back into a Drib message. When new messages are sent to the server application, they are assigned a message ID of 0. Thus if a new message is received, the 'addDrib' function assigns the message a unique ID using the DribbleIdentifier helper class discussed in section 9.4 and then adds it to the database using the dataset interface. An updateDrib method is also utilised in this class. This method is called when a user decides whether or not he or she likes a particular Drib and hence updates the Drib's like count. The method checks if a Drib is not null and hence updates the Drib's like count before adding the Drib to the database.

The EJB that is concerned with processing and returning a list of DribSubjects to the web tier is the 'getDribSubjectsBean'. As mentioned in section 8.4, a synchronous queue is established in the web service resource that requests a list of DribSubjects. In order to send DribSubjects back to the web-tier, this message-driven bean establishes a queue connection in it's constructor. The 'onMessage' method then creates a Queue object that connects the bean to the temporary queue as seen in *Figure 3*. Thus the web service resource that is waiting for a reply on the temporary queue, will receive the message from the bean. The parameters that are sent in a GET request as detailed in section 8.4 are received as message object properties in the bean. These properties are then stored in variables and using the 'getDribSubjects' method returns an ArrayList of the current most popular subjects. The getDribSubjects method initially retrieves all subjects that fall within the latitude and longitude of the user. Using the DribblePopularity class, the popularity of the subjects are calculated and the subjects are then sorted using the static 'rankSubjects' method.

Depending on the number of results requested, the ArrayList of subjects is then returned to the onMessage method and the ArrayList is then serialised and sent back to the web service resource using the temporary queue destination.

This bean also has a destructor called finalize that closes connections, destructs the bean as well as any resources associated with the bean [10]. This is needed as queue connections are limited and thus connections need to be closed as soon as a bean is not needed.

The 'GetDribsBean' is responsible for returning a list of Dribs associated with a particular subject. This implementation is very similar to that of the 'GetDribSubjectsBean'. However, in this case an additional GET query parameter called subject ID is retrieved in the bean as a message object property. This is necessary as Dribs need to be retrieved for a particular DribSubject with the above-mentioned subject ID. The getDribs method retrieves the ArrayList of messages from the database, ranks them in terms of their popularity and returns the list to the temporary queue destination. The bean possesses a destructor as detailed in the previous bean.

9.2 Popularity Score

The DribblePopularity class is responsible for calculating the popularity score of each Drib and DribSubjects. This class contains the functions 'rankSubjects' and 'rankDribs'. These are static functions as it is unnecessary to create a new popularity object for each instance of a bean object. The function rankSubjects calculates the popularity score of the DribSubjects. The subjects are received in an ArrayList and an iterator moves through the list calculating the popularity score of each subject. Subject popularity is defined in *Equation 1*.

$$DRank_{subj} = (1000(v) + 1000(p))(\frac{1}{2})^{\Delta x + \Delta t} \quad (1)$$

Here v and p are the number of views of a subject and posts to a subject respectively. Δx and Δt represents the location difference and time difference calculated in *Equation 2* and *Equation 3* respectively.

$$\Delta x = \frac{\sqrt{\delta latitude^2 + \delta longitude^2}}{8000} \quad (2)$$

$$\Delta t = \frac{t_{current} - t_{subject}}{30000} \quad (3)$$

$\delta latitude$ and $\delta longitude$ are the latitude and longitude differences between the user's current location and the DribSubject location. $t_{current}$ and $t_{subject}$ represent the current time and the last time the DribSubject was updated. The large constants are used to normalise the score.

The final $DRank_{subj}$ value is rounded up to the nearest integer and the subject popularity score is set. These equations allow for subject popularity to decrease by approximately half every kilometre, as distance from the original subject location increases. Time is also an important factor and popularity approximately halves every five minutes.

The Drib popularity is calculated using *Equation 4*. l represents the like count of the Drib and Δx and Δt represent the same parameters as in *Equation 2*. The Drib popularity also decrease with distance and time.

$$DRank_{msg} = (1000 + 1000(l))(\frac{1}{2})^{\Delta x + \Delta t} \quad (4)$$

The list of Dribs or DribSubjects are sorted in descending order of popularity and are returned to the beans.

9.3 Sorting

In order to efficiently sort the beans, Comparators are used and are implemented in the DribSubjectComparator and DribComparator classes respectively. A Comparator interface consists of a single method 'compare'. This method takes in two arguments and returns a 0, negative integer or positive integer [11]. This depends on whether the first argument is less than, equal to, or greater than the second argument [11]. The main objective of a comparator is to compare two different objects. It compare's other class' instances and sorts them using Collection.sort [12]. This algorithm sorts the Dribs or DribSubjects in descending order depending on their popularity score.

9.4 Unique ID

New Dribs and DribSubjects that are posted to the server are initially set with a message ID of 0. Thus the static function 'getUniqueID' is called from the DribbleIdentifier class in order to assign

unique ID values to the Drib and its DribSubject. The class implements the function 'getUniqueID' which is static as no more than one instance of DribbleIdentifier is needed. This function uses a HashSet to store the unique ID numbers. A Set cannot contain duplicate elements and is also considered to be a Collection [13]. The HashSet stores it's objects in a hash table. This is considered to be the best performing implementation but it does not guarantee the order of iteration through the table [13]. When a new number is generated using the Random object, the value is stored in the hash table if the value is unique. A 'deleteID' method is also necessary in this class as if Dribs or DribSubjects are removed, their unique ID must be removed from the hash table.

10. TESTING

10.1 Testing RESTful Web Services

One of the powerful features of using Glassfish and netbeans is the ability to initiate tests for RESTful web services from the netbeans IDE. The Test for RESTful web services allows the user to send PUT and GET requests to the URIs of the web service resources deployed on the application server. This also enabled the XML structure to be exposed that would be sent to the client. The XML structure that the client needed to send to the application in a PUT request could also be determined. This service enabled very efficient debugging and helped develop the standard communication protocol detailed in section 6.

10.2 Logging

The *java.util.logging.Logger* class is used as the main logging tool in Glassfish [14]. A $static final$Logger object had to be created and, using the getLogger method, bound itself to the relevant class. This object would then track all log messages reached during run-time. Two main log methods were used. $logger.info$ was used to provide general information during run-time and *logger.severe* was used to notify a user when an error had occurred. A typical log output is displayed in Appendix A *Figure A.4.* As can be seen the time the message was produced is tabulated. The type of message is displayed which in this case is an INFO message. The class in which the message was logged is also displayed as well as the message itself. It can be seen from this setup that a very efficient system exists for debugging the application.

11. TIME MANAGEMENT

Developing the application involved learning many new types of technologies. Thus making estimates on the expected development time could have been very inaccurate. The estimated development time for the overall project as seen in the comprehensive project time sheet in *Table A.1* was 110 hours. The estimate turned out to be 23 hours too short. This was expected for the above-mentioned reasons.

12. CRITICAL ANALYSIS

12.1 Trade-offs

When calculating the range around a location to which Dribs and DribSubjects are sent, the range was approximated as a square around the user. This made for simpler calculations. When a user chooses to like or dislike a Drib, the entire message is returned to the application server with the like/dislike value. This provided simpler implementation but created extra overhead. A neural network was to be implemented to determine the popularity of Dribs and DribSubjects. This created far too much complexity for the given time frame and thus a ranking system was adopted.

12.2 Recommendations

In developing server applications of a similar nature in the future, it is recommended to create a feature in the application to periodically delete messages off the database. The current application will simply scale as more messages get added to it's database. Since no messages are ever deleted, the server may ultimately fail. Load balancing may be implemented with redundancy by added extra servers to run the same application and spreading the load among the different servers. This can also be used to implement fault tolerance within the application. It may be beneficial to decouple the dataset object from the message-driven beans using queues. Thus by implementing synchronous queueing between the beans and the dataset, the two layers are effectively separated and this may prove to be more efficient and modular. Unit tests could be written

for the application to create an effective testing package. It is also recommended to load test the server using programs such as JMeter and SoapUI. If the application is going to be marketed commercially, then the application needs to be exposed to real world scenarios.

13. CONCLUSION

A server-side application has been developed that integrates with the client application to form a location-based social messaging network. The server application was implemented using the JEE framework. The application was deployed and managed on the Glassfish application server. The application itself consists of a web component, a beans component and a data component. The RESTful web services allowed for an object-oriented approach to developing a standardised communication protocol. The beans component consists of the message-driven beans that wait asynchronously for objects on JMS queues. The objects are then processing using methods within the beans as well as a variety of helper classes. Unique IDs and popularity scores have been implemented as well as sorting algorithms to send Dribs and DribSubjects to the clients in order of popularity. Testing of RESTful web services as well as logging helped in the debugging of the application. The time management of the project has been detailed with a comprehensive project time sheet. Application trade-offs as well as recommendations for similar applications of the same nature have been detailed.

REFERENCES

[1] *JEE framework*. http://www.exforsys.com/tutorials /j2ee/j2ee-overview.html, Last accessed 30 April 2010.

[2] *JDBC Resources*. http://docs.sun.com/source /819-0215/jdbc.html, Last accessed 02 may 2010.

[3] Sun Microsystems. *RESTful Web Services Developers Guide*, 2009.

[4] *Using Annotations*. http://today.java.net/pub/a/ today/2007/05/22/using-annotations-in-java-ee-5.html, Last accessed 02 May 2010.

[5] *UriInfo JavaEE 6*. http://java.sun.com/javaee/6/docs/api/javax/ws/rs/core/UriInfo.html, Last accessed 02 May 2010.

[6] *JMS Overview*. http://java.sun.com/products /jms/tutorial/1_3_1fcs/ doc/overview.html#1027335, Last Accessed 29 April 2010.

[7] *J2EE JMS*. http://java.sun.com/j2ee/sdk_1.3 /techdocs/api/javax/jms/package-summary.html, Last accessed 02 May 2010.

[8] *Introducing the context*. http://docstore.mik.ua/orelly/java-ent/jenut/ch06_03.htm, Last accessed 02 May 2010.

[9] *Message-driven Beans*. http://www.redhat.com/docs/manuals /jboss/jbosseap4.2/doc/ Server_Configuration_Guide/EJB3_Services Message_ Driven_ Beans.html, Last accessed 30 April 2010.

[10] *Java Quick Reference*. http://www.janeg.ca/ scjp/gc/finalize.html, Last accessed 02 May 2010.

[11] *Object Ordering*. http://java.sun.com/docs/books/tutorial /collections/interfaces/order.html , Last accessed 02 May 2010.

[12] *Java Sorting*. http://lkamal.blogspot.com/2008/07/ javasortingcomparatorvscomparable.html , Last accessed 02 May 2010.

[13] *The Set Interface*. http://java.sun.com/docs/books/tutorial /collections/interfaces/set.html% Last accessed 02 May 2010.

[14] *FaqConfigureLogging*. http://wiki.glassfish.java.net /Wiki.jsp?page=FaqConfigureLogging, Last accessed 02 May 2010.

Appendix

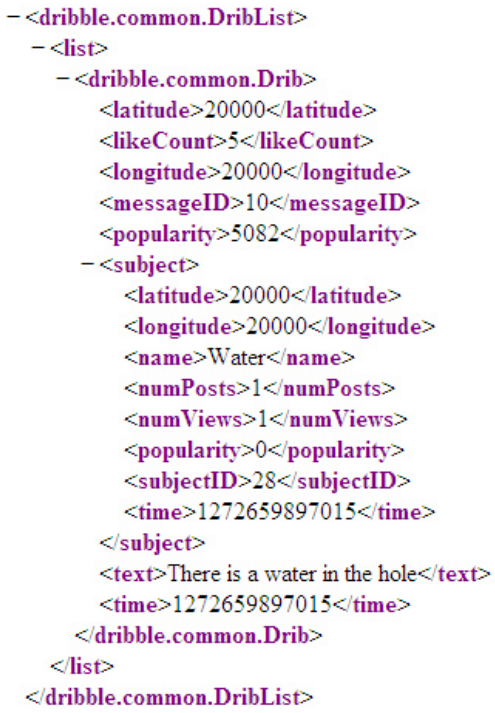


Figure A.1 : Typical XML Drib Structure

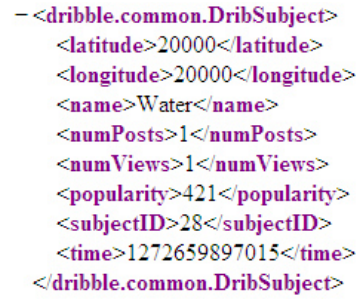


Figure A.2 : Typical XML DribSubject Structure

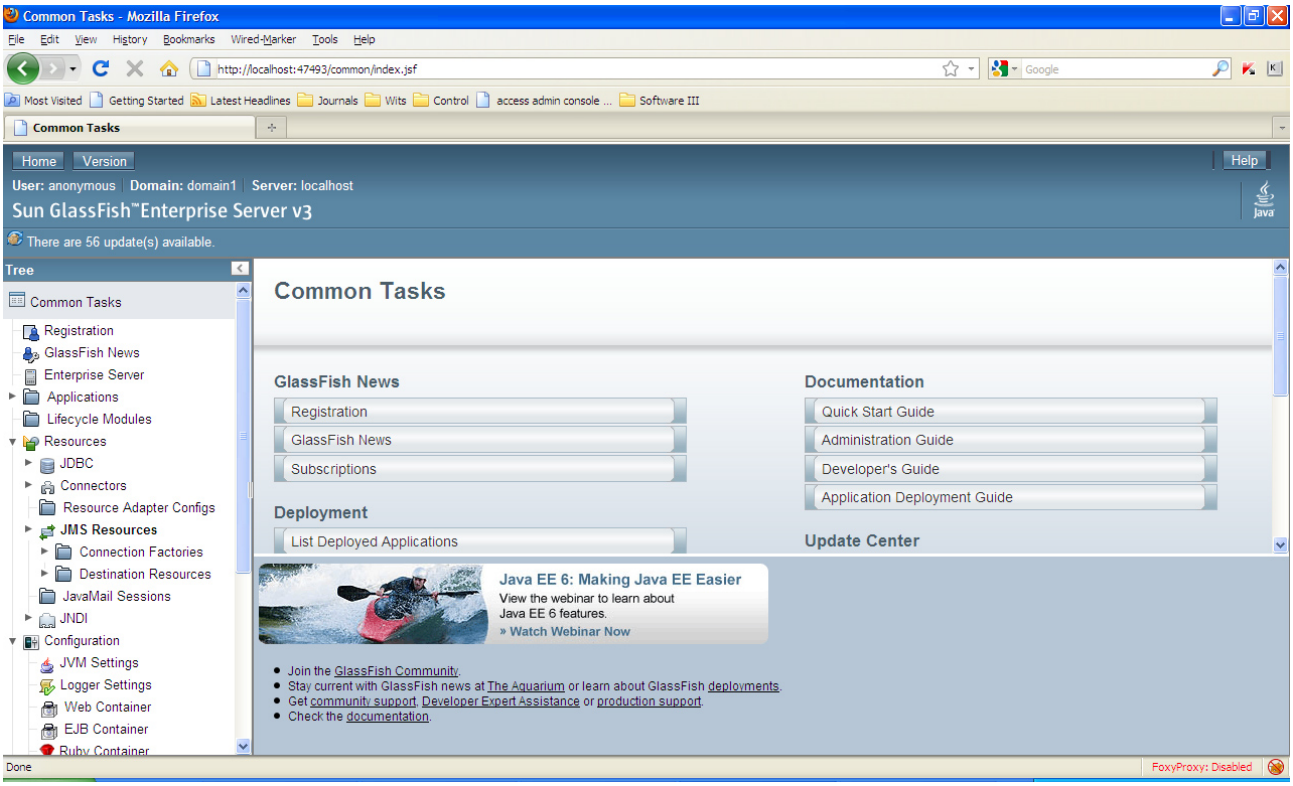


Figure A.3 : The administrator console of the Glassfish application server

```
3331
3332  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|SQLCommunicator|_ThreadID=19;_ThreadName=Thread-1;|Subject Found|#]
3333
3334  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|SQLCommunicator|_ThreadID=19;_ThreadName=Thread-1;|Getting the requested DribSubject|#]
3335
3336  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|SQLCommunicator|_ThreadID=19;_ThreadName=Thread-1;|Added name field to subject object|#]
3337
3338  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|SQLCommunicator|_ThreadID=19;_ThreadName=Thread-1;|DribSubject populated... Returning DribSubject|#]
3339
3340  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|GetDribSubjectsBean|_ThreadID=19;_ThreadName=Thread-1;|Calculating popularity scores|#]
3341
3342  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|GetDribSubjectsBean|_ThreadID=19;_ThreadName=Thread-1;|Sending response|#]
3343
3344  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|GetDribSubjectsResource|_ThreadID=31;_ThreadName=Thread-1;|Response received|#]
3345
3346  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|GetDribSubjectsResource|_ThreadID=31;_ThreadName=Thread-1;|Wrapping list of subjects|#]
3347
3348  [#|2010-04-30T22:49:31.921+0200|INFO|glassfishv3.0|GetDribSubjectsResource|_ThreadID=31;_ThreadName=Thread-1;|===== DribSubjects sent to client =====|#]
```

Figure A.4 : A log output for sending a DribSubject to the client

Table A.1 : The comprehensive project timesheet

Sub-task	**Estimated time(hours)**	**Actual time (hours)**
Server-side		
Setting up Web-server	10	12
Webservices	16	18
Synchronous Queues	2	3
Queues	6	10
Enterprise Beans	16	20
Processing Algorthms	8	8
Dataset Interface	4	6
Database Implementation	16	20
Integration	16	18
Testing	16	18
Total	110	133