Offline Database Synchronization using SOAP and MySQL

# Matt Nesterenko

# CS526 – Advanced Internet and Web Systems

# Research Project Report

# Spring 2012

# Dr. Edward Chow

# Abstract

Many mobile applications today lack support when the user goes offline. This project, called Offline Database Synchronization using SOAP, aims to provide a basis to aid in the development of software which is capable of operating offline and synchronizing with a server. The communication between the client and the server is sent using SOAP. Changes are noted on both the server and client using a method called data auditing to aid in the synchronization notifications between the client and the server. Future additions to be made in real-world examples would largely include security implementations as this project focuses only on synchronization – not security.

# Introduction

Over the years, the industry for mobile computing has been continually growing. One growing trend is the use of client-server applications which require internet access to save and retrieve data. While this is great for reducing an application’s code footprint and placing the processing load on a dedicated server, it ultimately suffers from one issue: what happens if the device goes offline? From the moment this occurs, the program will be useless to the user as the data source for the application is unreachable. One possible solution would be offline data synchronization. This method allows the user to continually work with their application should they go offline. When the user comes back online, the application simply resynchronizes data with the server and the user will be able to continue their work.

This application acts a simple task list. The client can add, edit, and delete tasks locally and synchronize with the server at any time. Other clients can synchronize with the server as well, as the database sync method supports multiple users modifying the database at once. The goal of this project is to demonstrate a simple application which supports offline data modification and synchronization.

The project this paper will cover deals with the design and implementation of such a solution. There are many ways which this can be implemented; however, this project focuses on a few key technologies to allow for a greater portability and standardization. The primary language used is Java, and the data is stored using the MySQL open source database. Communication between the client (simulated mobile device) and server uses Simple Object Access Protocol (SOAP) over HTTP. These will be discussed in detail along with the outcome of the project and possible steps which should be implemented in future and real-world developments.

# Database Synchronization

The majority of this project deals with how the data is actually synchronized in the database. There are a few approaches in which a program can accomplish this. Tradeoffs between each of the methods are their complexity and the benefit each has to offer. A few basic options will be discussed as well as the method used in this project.

## Basic Database Synchronization Techniques

The easiest synchronization method would be a simple database flush. Once the client device is online and initiates synchronization, all of the data on the server is flushed out and replaced with the data from the client. This method is extremely simple, but suffers from heavy drawbacks. Changes are sent one way (client to server), preventing multiple users from using the system. In addition to the lack of multi-user support, it is very bandwidth intensive as all of the data on the client is resent to the server during each synchronization. This can be easily prevented by sending only modified records.

Instead of flushing and uploading all data, the next step up would be comparing data on each side. This involves checking modifications on all of the data on both the client and the server. This would effectively prevent the client from uploading all the data to the server; however, now the issue is that there is a massive amount of computational power involved in comparing this data. The benefit from using this method is that multiple users can use the system, only specific information that each user modified can be overwritten. Despite this multi-user advantage, it is more than arguable to say that the overhead cost outweighs the benefit. One may suggest using a synchronization flag to mark the data which needs to be synchronized. This would be much more efficient, but the multi-user support is no longer available.

## Data Auditing

The method of synchronization this project utilizes is called Data Auditing. There are no flags for marking records with that indicate a necessary synchronization. Instead, a data table is used for noting changes that have been made. For example, say the client creates a new record. Two records are created. The first record is created in the data table where all application data resides. The second is placed in the audit table. The audit record contains a link pointing to the data record as well as what action was performed (in this case, indicating data creation). The same auditing goes for data modification and deletion. If a record is edited, an audit record is created indicating such is created. When an item is deleted, the ID of that item is preserved even though the data record is deleted. This ID is used for identifying which record to delete on the client.

Multi-user support can be added to this system by adding an identifier to the audit log to represent some unique value associated with each user on the system. The project opts for a simple integer device id to differentiate between users. This allows a client to request all the modifications made on the server from all devices IDs excluding the local ID.



The client must also know where the last synchronization finished. There are two main solutions for this. The first allows the client to request all updates from the server since the last time it synchronized. The server relays this information by finding the last update by requested device ID, and then retrieving all records in the audit log after that item. The second method requires the client to save the last update received from the server. If changes are not committed in one database transaction, it may be useful to choose the latter option to avoid synchronization issues if multiple users are synchronizing simultaneously.

# Design

Like mentioned before, this project utilizes three core technologies: Java, MySQL, and web services with SOAP. Java was chosen for the portability of code on devices which support the JVM. Despite virtualized code obviously being slower than native code, the performance impact will be negligible when compared to network latencies.

MySQL is an extremely popular open-source database application. It was chosen for this project because it meets the basic needs of this project’s data scheme. In addition, NetBeans has support for MySQL built-in with the EclipseLink 2 data Persistence manager. EclipseLink is not essential for this project; it is used for generating data transactions as well as reading data from the database with reduced code.

The client communicates to the server through a web service. In a nutshell, a web service is like a class with methods which can be called online. The web service communicates using SOAP, which is XML encoded data. The format of this data can be located in the web service’s Web Service Description Language (WSDL) file. All web services must provide a WSDL to ensure error-free communication with clients. Using a web service for data synchronization is useful because it is an open protocol, and does not utilize any proprietary technologies. Devices would not be required to be written on a specific platform or use a commercial language to communicate with web services.

## MySQL Schema

The table schema for the task synchronizer is fairly simple. All integer ID numbers are automatically incremented by MySQL with the use of the auto\_increment flag. Tasks only contain a name, so they are stored in the database as a varchar(32) with an additional ID field. Since auto\_increment is used, ID numbers will not match client and server side. To handle this issue, the ClientTask table contains an additional integer field called “svrid.” This field is used to map back to the ID of the same task on the server. The server-side audit table contains an integer “devid” field, which contains the ID of the device which requested the change. A full table listing can be seen below.

A small table called ClientInfo exists on the client to store synchronization data. This table contains a field with the name “lastsync” and an integer value containing the last audit-ID retrieved from the server. This field will be used to retrieve future audits and will be updated with such.

All audit tables contain a “mode” field. This is an integer value which represents what type of data modification occurred. The value “1” signifies data creation, a “2” means data was edited, and “3” stands for deletion.

## Web-Service Methods

In this project, the client device handles all synchronization features. The server is only responsible for data access and audit access. The client accesses these features by accessing five web service methods. The first three are straightforward: *AddTask(string name, int devid*)*, EditTask(string name, string newtask, int devid)*, and *DeleteTask(int tasked, int devid)*. Each of these methods manipulate the server-side database and audit the changes. A method named *GetTask(int taskid)* exists to retrieve a task with a specific ID. Below is an example of the SOAP messages sent and received for the AddTask method.

**Example SOAP Response:**

<?xml version="1.0" encoding="UTF-8"?>

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">

<S:Body>

<ns2:AddTaskResponse xmlns:ns2="http://webservice/">

<return>6</return>

</ns2:AddTaskResponse>

</S:Body>

</S:Envelope>

**Example SOAP Request:**

<?xml version="1.0" encoding="UTF-8"?>

<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">

<S:Header/>

<S:Body>

<ns2:AddTask xmlns:ns2="http://webservice/">

<name>A Task</name>

<devid>55</devid>

</ns2:AddTask>

</S:Body>

</S:Envelope>

The last method is named *GetAudits(int devid, int lastupdate)*. This method returns a list of audits from the ServerAudit table excluding audits from the “devid” parameter, and only returning updates after the “lastupdate” parameter. This list only contains the IDs of the tasks modified. GetTask() must be called to retrieve the task itself to update the client-side database.

# Implementation

Web services are not run like normal java applications, such as a console or GUI window. To use the web-service, it must be deployed to a web-server using an application such as Apache Tomcat, GlassFish, or JBOSS. GlassFish was chosen as the server host application for this project as it was preconfigured with NetBeans. After compiling the server, the only steps necessary to deploy the application are right clicking the project, and clicking the “Deploy” option. This compiles the code into an application file which is readable by several Java host applications.

The MySQL version for this project is 5.5.16, installed as part of XAMPP. The tables were created in a single database and then accessed from within the Java client and server. Despite the tables being in the same database, the server never accesses the client tables, and the client never accesses the server tables. This was only done for simplicity, as a real world example would have separate databases and servers for the client and server.

The client is a simple console application, which displays a menu with basic data manipulation options. All data on the client is modified locally and the user must explicitly request a synchronization to update data on both ends. No special installation is necessary for the client as it can be run directly within NetBeans.

All database usage is done from within the EclipseLink persistence manager. This acts as a layer on top of a database, allowing classes to represent data within a table. Editing and data is a matter of creating a transaction, modifying the class variables, and then committing the transaction. Since the synchronizations are transaction based, all changes are committed when the synchronizations are complete. Using a persistence manager creates a larger database access overhead. In data and time intensive applications it may be wise to use raw SQL queries to optimize query execution time.

# Challenges

One of the primary challenges faced in this project was connecting the web service to a static persistence unit. The persistence unit allows the program to create and finalize transactions and manipulate data. Creating persistence units is a very time costly event as there is a large data-transfer from the database to the server. The solution is to use a Java feature called bean injection. This operation makes the web service create a persistence unit as soon as it is initialized. Every database operation following will use that same unit, instead of creating a new one. The syntax for this injection:

//inject the persistence unit factory

@PersistenceUnit private EntityManagerFactory emf;

//inject the transaction for the persistence unit

@Resource UserTransaction ut;

From the injection, one may simply call *emf.CreateEntityManager()* to retrieve the entity manager and use it in a much more timely manner. Transactions are handled by using the injected UserTransaction class. A transaction is started by calling *ut.begin()* and finalized by calling *ut.commit()*.

Another challenging aspect of this project was the side-effects of using auto-generated IDs for the database. This led to ID fields being out of sync between the server and client. For example, Client A adds 3 records while offline. During this time, Client B synchronizes with the server, adding 2 records. When Client A initiates a sync, the IDs of the records added to the server will be two greater than on the client due to Client B adding the records. Now a problem exists, the two items on Client A’s phone do not match the IDs of the same two items on the server. The solution implemented is to store the server IDs client side. When the client modifies data on the server, it will use the ID saved for the server to indicate which record will be changed. The client obtains the ID on the server when an item is added; the SOAP response contains an integer of the ID or -1 if the add failed.

# Lessons Learnt / Future Direction

This project demonstrates how to create a simplistic server-client model which supports offline data manipulation and synchronization using SOAP. This project, being as basic as it is, has several opportunities for improvement.

Time constraints prevent any sort of security implementations to be made regarding this project. In a real-world scenario, security would be added to many areas of this scenario. For example, there is no authentication service for synchronizing. Any user could submit soap requests to the server and manipulate data with ease. In addition to this, the data sent between the client and server is unencrypted. Any sensitive information can get sniffed from the network and read in plaintext. A few solutions for this issue exist.

One method of securing transferred data is to use HTTPS (HTTP Secure). This protocol encrypts data at the transport layer (layer 4) of the OSI network model. This method is commonly used for web sites which require high security. A public key infrastructure (PKI) is commonly used with TLS. Because SOAP is not required to run over a specific protocol, it is not too difficult to implement data transfer over HTTPS instead of the defaulted HTTP. The downside to using HTTPS is that the data is decrypted before it reaches the application. If client or server security has been breached, it would not be difficult to intercept network data flow after the packet has been decrypted. Some prefer to use a security implementation called WS-Security.

Should an application require more security that HTTPS can offer, a SOAP security extension exists. WS-Security focuses on application layer security more than transport layer. Information for various token models can be sent in the SOAP headers, while encrypted data resides in the SOAP body. WS-Security supports various specifications such as: X.509, Kerberos, UserID/Passwords, as well as other custom-defined tokens. For X.509, the certificates are sent inside of XML tags, along with digest information. Information is not decrypted until after it has reached the process requesting the data. If the document is reaching many sources and only specific information should be available at each, multiple certificates are sent and only computers with the corresponding keys can decrypt the information. The only issue with using WS-Security is the significant slowdown factor. Unless WS-Security is essential, it may be worth the performance gain to use HTTPS/TLS. Wikipedia reports the following benchmarked security statistics:

|  |  |
| --- | --- |
| **Security Mechanism** | **Messages/second** |
| WS-Security (X.509) XML Signature & Encryption | 352 |
| WS-SecureConversation XML Signature & Encryption | 798 |
| Transport Layer Security | 2918 |

In addition to implementing security, the data transferred over SOAP is raw data from the database. To make an elegant design, creating classes which contain a higher human-readable element would increase overall code readability. When audit information is sent to the client from the server, it only contains task IDs which must be retrieved from the server later in time. To make this more efficient, creating classes which contain audit information, IDs, mappings, and data would decrease overall bandwidth and requests. This would slightly increase the code footprint, but reduce bandwidth and processing time.

# Conclusion

Many mobile applications today do not yet support going offline. Although some applications do require a persistent online connection, many could implement a form of offline-synchronization to reduce limitations on the user. The offline database synchronization project demonstrates a small example of how one can achieve this. Developers may choose not to use Java or web services to accomplish this task, however, many will choose the same style of synchronization: data auditing. This style has the power of performing an entire database comparison without the overhead of matching every data row. Instead only the changes are kept and forwarded to other devices. Only time will tell if offline database synchronization will become a trend in mobile application development.

# References

*SOAP*. (2012, April 19). Retrieved May 1, 2012, from Wikipedia: http://en.wikipedia.org/wiki/SOAP

*WS-Security*. (2012, March 31). Retrieved May 2, 2012, from Wikipedia: http://en.wikipedia.org/wiki/WS-Security

Bertocci, V. (2005, April 25). *End to End security, or why you shouldn't drive your motorcycle naked*. Retrieved May 2, 2012, from MSDN Blogs: http://blogs.msdn.com/b/vbertocci/archive/2005/04/25/end-to-end-security-or-why-you-shouldn-t-drive-your-motorcycle-naked.aspx

Nadel, B. (2010, September 7). *Experimenting With Offline Data Synchronization Using jQuery And ColdFusion*. Retrieved April 2, 2012, from The Blog of Ben Nadel: http://www.bennadel.com/blog/2005-Experimenting-With-Offline-Data-Synchronization-Using-jQuery-And-ColdFusion.htm

Powell, M. (2001, November 21). *Real SOAP Security*. Retrieved May 2, 2012, from MSDN: http://msdn.microsoft.com/en-us/library/aa480522.aspx

Appendix

# Installation

### MySQL Configuration

An easy way to create the data tables and insert the correct information into the ClientInfo table is to use the CreateTables.sql file included with the source code. The relative location of this file is $PROJECT\_PATH/server/src/java/CreateTables.sql. If using multiple databases for this project, one can separate the client and server tables to their designated databases.

### Server/Client Configuration

Persistence units are used in both the server and the client Java code. The JDBC connections of each will need to be changed to suite the database credentials for each unit. The persistence.xml file for the client can be accessed through NetBeans in the path client -> Source Packages -> META-INF -> persistence.xml. The server persistence file is found in the path server -> Configuration Files -> persistence.xml.

Once these files are modified, one must deploy the server before it can be accessed by the client. Assuming GlassFish is installed, right click the server project and click Deploy. Once this completes the client can be run by right clicking the client project and clicking Run. The client output should appear in a built-in NetBeans console window or an external one; depending on the configuration of the IDE.