

Updateable SQL Plugin

Wirox

Eugene Marcotte
Chris Daniels
Paul Yates
Matt Blackwood

Xerox

Peter Alfvin
Timothy Luksha

Faculty Mentor

Professor J. Scott Hawker

Project Overview

Xerox uses Confluence Wiki to help organize their project related communications and information artifacts. They use Bob Swift's SQL plugin to embed external database information in wiki pages as well as generating data for charting on the wiki. Currently developers use a separate tool to manipulate the database tables and views. They wanted the plugin to allow them to edit, insert, and delete database information through wiki pages. Xerox asked us to develop an extension for the plugin which would provide them with the basic functionality to perform row operations on queries that are embedded in wiki pages. Time permitting the project would also include multi-row operations, advanced data type support, and table scrolling. Another major feature, time permitting, was user-specific database sessions. Currently the plugin uses a single identity provided in a configuration file when performing database operations. The user-specific sessions would allow the plugin to connect the user's identity with the operations they perform. The extension to the plugin needed to be written in Java. It was intended to work with MySQL and Oracle 10g database servers. It needed work in both Internet Explorer and Firefox browsers on Windows XP and Firefox on Solaris.

This project took place over a twenty week development period in which incremental deliverables were expected by the customer. Utilizing a combination of XP and Scrum methodologies a maximum of two week sprints was used to deliver each release. Frequent communication and stakeholder accessibility was paramount in running successful sprints and allowing for quick response to project issues.

There was a Scrum Master that rotated every six weeks throughout the project. The responsibilities of the Scrum Master included running the daily scrum meetings (fifteen minute long meetings to communicate status and impediments), work with the product owners to further refine requirements, and help remove any impediments the team had. The Scrum Master was also accountable for updating story point burn-down charts and tracking metrics throughout the sprint as well as updating the team time sheets.

Since we were using test-driven development, team members were responsible for writing tests for the functionality they added.

The senior project team was responsible for choosing which user stories were implemented in each sprint based on priority (determined by the Product Owner), as well as story size and velocity (determined by the team). Team Wirox was also required to track progress as well as keep time cards of weekly tasks and achievements.
Basic Requirements

The basic requirements of the editable SQL plugin were based around providing a way to perform insert, update and delete operations on SQL query results displayed within a wiki page. The system would also require some level of simple security and also allow the ability to do multiple row operations (multiple updates in one transaction). The user dictates the SQL query within the plugin macro and our responsibility is to present the results of this query and allow the user to manipulate the returned data. User stories are classified under updating, inserting, deleting, and user interface.

Updating:

The updating class of user stories was quite large. The first thing a user needed to be able to do was to refresh the result set from an SQL query. This would allow the user to see the most up to date data in whatever table or tables they were viewing. The refresh needed to happen in an AJAX like way, such that the entire web page needed no refreshing. Next users needed the ability perform a row edit and then save it. This would allow the user to make updates to a row in a query. This editing of rows needed to happen in tables that had primary keys and tables that did not. For columns that are nullable the plugin needed to provide a way for users to input null values. The next requirement Xerox specified that went along editing rows was the ability to do multiple row edits and to save them all at once. Multiple row transactions was a very important requirement because it gave the user the ability to make edits as they go and then save all at once, instead of each time they made an edit. Users also needed to have the ability to cancel both a single row edit and multiple row edits. This would allow the user to bring back the original values before a commit takes place. The plugin had to provide support for several basic data types. Our sponsor Xerox dictated that they wanted support for varchars, dates, numbers, and timestamps in oracle. We also provided support for integers, and decimals in MySQL. Our solution also provides validators for columns to make sure that users are inputting the correct format for numbers as well as the correct input type for columns (i.e. no varchars in number inputs).

Inserting:

Users needed the ability to insert rows into a given table and the plugin needed to provide a quick and intuitive way to do this. The insertion needed to be able to done with all previously mentioned data tables as well as provide the user with the ability to add null values to nullable columns. Inserting also needed to have validation on a user’s input before it took place. Xerox required that we either tack on an extra row to the table or provide the user with a dialog from which they could enter the data for a new row.

Deleting:

Users must be able to delete a row from the table. This deletion must involve a confirmation step before proceeding. If a user decides to cancel that deletion, then no deletion must take place. Maintaining database integrity was very important to Xerox, due to the fact that this plugin would be used on their primary databases.

User Interface:

Due to the environment our plugin would be working in, Xerox asked that we provide support for both Firefox and Internet Explorer browsers on both Windows and Solaris machines. The plugin also had to provide some sort of security. For security our solution relies on Confluence’s user permissions on a wiki page to either give read only access or write access. When selecting a table initially our solution only allows select statements to prevent the use of malicious SQL statements. Finally Xerox also asked for some sort of scrolling or paging for large tables. For large tables in our solution we have provided the option to page them.

Constraints

There was a separation between constraints that are project specific and department specific. The difference lies in that some constraints were determined by the specifications we recieved from our project sponsor and others are determined by SE department requirements.

Sponsor Constraints

- Atlassian Plugin Framework - Since we are developing a plugin with the intention of it being available to all users of Confluence (not just Xerox) we need to follow the Atlassian Plugin Framework in order to have Atlassian accept our project as an offered plugin. The framework allows
for adding of Servlets and macros to a Confluence instance, including Maven project archetypes and tasks.

- Java - The original plugin was written in Java and so we had to continue in this regard. Confluence is developed on Java technology and allows plugins to register Servlets to perform desired functionality.

- Oracle - Oracle is the database in use by Xerox. It was critical that support for Oracle be available but it was also important that we put in an effort to make the plugin as independent as possible of the underlying database. We needed to test the required data types in Oracle first to ensure that our sponsor would be able to make use of the system that was delivered.

- MySQL - MySQL was not an explicit constraint but we began our development on MySQL thinking it would translate well to Oracle. Keeping in mind that we wanted our plugin to be useful to more than Xerox we maintained tests for MySQL and tried to make available the same functionality that is in Oracle.

- Maven - Maven is the build platform used to manage dependencies and automate many Confluence development tasks.

- Budget - Since this is a senior project, and was intended to be open source, we had no budget and had to rely on open source libraries.

### Software Engineering Department Constraints

- Development Time - The SE department expected a minimum of ten hours of work each week. We generally put in about fifteen hours of work each week. A tracking report for time/effort worked by each team member and the team aggregate had to be updated on the project website weekly. We were restricted to a twenty week development period.

- Website - The SE department also mandated that we maintained a project website holding all non-proprietary work products and project artifacts.

### Development Process

The process methodology that the team used for developing the SQL plugin system was Scrum with XP engineering. This was chosen as a result of previous experience within the team, a desire for iterative development and the fact that Xerox uses the same methodology. This would allow us to be very flexible with requirements and schedule changes and there would also be a greater understanding between the team and sponsor about the process used.

Communication with the sponsor was continuous and open. We had planning meetings for our sprints every two weeks and there was also correspondence in between meetings to clarify questions about user stories or to notify the sponsors of issues. Meetings were held at RIT with an option to attend remotely using AdobeConnect and/or telephone and were held to establish the progress of the previous sprint and determine the goal and user stories for the next. E-mail was used in between sprint meetings in order to keep in contact and keep everyone up-to-date.

Defect tracking was performed using an issue log on our project's wiki. This issue log recorded the impact, priority, assignment, and comments regarding defects in our plugin. These issues were added to the sprint backlog based on priority of the issue and the current status of the sprint backlog items.

The team initially had a set of roles that would be rotated within the team every sprint. These were made up of tech lead, scrum master, configuration manager, and test lead. We arranged these so that everyone in the group would have a responsibility for each sprint. These did not work well and we decided it is better to have low role overhead and just use the scrum master in line with Scrum practice. The scrum master role was then rotated every three sprints to maintain some level of consistency within the role. The scrum master was primarily responsible for communicating with the project sponsor in between sprint meetings and also help to remove serious impediments when they occurred. The other roles were just collapsed into a collaborative effort to keep the project moving, simply using the collective intelligence of the team to troubleshoot problems and maintain velocity.
Project Schedule: Planned and Actual

There were two somewhat disjoint schedules for this project. When we developed or project plan we knew the time we had for project completion and also the mandated milestones by the SE department. This helped us develop a timeline that showed milestones and the start and end dates of our separate sprints. Although the sprints were planned within the administrative schedule, their content was not. We would check this schedule periodically to make sure that we were fulfilling non-sponsor related responsibilities.

The main administrative schedule tasks were related to presentation dates, required project artifacts, and updates on our project progress. These were helpful on a macro project level but for development we used sprint planning meetings. These sprint planning meetings determined the user stories to be completed for the following sprint and thus provided us with a schedule for that period of time. The user stories were broken down into tasks and this determined our day-to-day activities. The sprint planning was relatively loose since we needed to be flexible in order to be responsive to impediments or possible changes in the requirements.

There were not many discrepancies with the actual plan compared to the original plan. We met deadlines for the administrative schedule and the sprints did not contain rigid scheduling. Within sprints we had the ability to add or backburner user stories in order to tackle the amount of work that was allowed by our schedules and the impediments that were encountered. We would adjust sprints to reflect challenges or increased momentum and in the end were able to complete all user stories within our project schedule.

System Design

Server side

Initial Design Approach

Our first technology spikes, prior to starting our first sprint, involved modifying Bob Swift's SQL Plugin to display query results in an HTML form. For the server-side technology spike, the output of the SQL Plugin was modified to return a static HTML form containing input fields for each cell of the query results. To handle processing changes to the query results, a Servlet was written that would accept an HTTP POST request containing the SQL query and the form values with updates. With these parameters, the Servlet would then create and populate a ResultSet, navigate to each row in the ResultSet, call column update methods on each column for the row and invoke the updateRow method to update the current row of the ResultSet. After all row modifications were made, the changes were commited by the ResultSet via the acceptChanges method.
First Approach - Creating update statements using JDBC ResultSet Metadata, Javascript sends limited POST data (table id, original values, new values)

During our first sprint, we continued to prototype and spike technologies to mitigate project risk. For sprint 1, this involved prototyping AJAX and Web 2.0 features provided by JavaScript libraries bundled with Confluence (http://confluence.atlassian.com/display/DOC/Including+Javascript+and+CSS+resources#IncludingJavascriptandCSSresources-IncludingaJavaScriptLibraryprovidedbyConfluence). Using jQuery, we were able to send AJAX requests containing only the rows that were updated.

After spiking the ResultSet using ResultSet.CONCUR_UPDATABLE and discovering limitations (http://www.mcs.csueastbay.edu/support/oracle/doc/10.2/java.102/b14355/resltset.htm#CIHEJHJI) including:

- limited support for SELECT * queries (in cases all columns needed to be specified)
- only supporting updates to one underlying table of the SELECT statement
- inserting values only when the SELECT statement includes all non-nullable columns

At this time, we developed a design that relied on ResultSet metadata to craft specific SQL UPDATE statements. In this way, we would be able to update to multiple underlying tables for SELECT statements used in joins and views. The approach involved:

1. Mapping each column name to its underlying table name using getTableName provided by ResultSetMetaData (http://java.sun.com/j2se/1.4.2/docs/api/java/sql/ResultSetMetaData.html)
2. Creating a set of distinct table names
3. Mapping each table name to a collection of column update values
4. Using these two maps to create SQL UPDATE statements
5. Executing these statements
Second Approach - Creating update statements using JDBC ResultSet Metadata & additional metadata retrieval code, Javascript sends limited POST data (table id, original values, new values)

Unfortunately, the Oracle 10g JDBC driver does not implement the getTableName method correctly and returns an empty string "because [the] underlying protocol does not make this feasible" (http://download.oracle.com/docs/cd/B14117_01/java.101/b10979/datacc.htm#sthref914); Database specific workarounds were investigated that involved querying system tables containing metadata such as table name, column names, etc. to allow our first design to work for Oracle datasources, but these approaches were not feasible.

Third (Final) Approach - WebRowSet serialization to XML, client-side processing of XML with jQuery

The final design for the server-side components of the Updateable SQL Plugin uses RowSet technology. WebRowSet is an interface introduced in JSR 114 and Java 1.5 that allows for the serialization of a RowSet to XML. WebRowSetImpl is a Sun implementation of WebRowSet interface. A RowSet is a disconnected result set in the sense that it is possible to retain and perform operations on the result set without an open connection to the underlying database. This allows a RowSet to be modified and have changes committed to the underlying database. For the purposes of our plugin, the result set returned by executing the SQL query provided to the macro is serialized to XML using the WebRowSet. This XML data is then transported to Confluence users via HTTP. By using result set serialization, the server-side components became more cohesive and simplified. Our POJOs were no longer responsible for crafting appropriate SQL statements corresponding to plugin user row operations (update, insert, delete) as this functionality is provided by the WebRowSet implementation. The RefreshServlet is responsible for executing the macro provided SQL statement, creating a WebRowSet, serializing the WebRowSet to XML and writing this XML data to the servlet output stream. AbstractServlet provided a template-method pattern for including shared HTTP request parameter checking and session authentication checking steps prior to invoking the main primary method of the Servlets -- the execute method.

With this approach, the Refresh servlet essentially was a few lines of code:

```java
PrintWriter out = resp.getWriter();
String sql = req.getParameter("sql");
try {
    WiroxWebRowSet wrs = null;
    wrs = new WiroxWebRowSet();
    wrs.setDataSourceName("java:/comp/env/jdbc/" + req.getParameter("datasource"));
    wrs.setCommand(sql);
    wrs.execute();
    wrs.writeXml(out);
    wrs.close();
} catch (SQLException wrse) { 
    out.write(getErrorXML(wrse));
}
```

The UpdateServlet is responsible for accepting the XML from the client-side containing any row operations and new data. Row updates are captured by including a updateRow element providing the updated value of a column of a row (as shown below).

```xml
<row updateRow=
    <columnValue>3</columnValue>
    <columnValue>Matt</columnValue>
    <updateRow>Matt B</updateRow>
    <columnValue>Goshen, NH</columnValue>
</row>
```

An inserted row is captured by a insertRow element with columnValue child elements representing the values of the row to be inserted.
A row delete operation is captured by a deleteRow element with columnValue child elements.

```
<row delete operation.  Note the second column is a timestamp. -->
<deleteRow>
  <columnValue>3</columnValue>
  <columnValue>1577854800000</columnValue>
</deleteRow>
```

A column can be specified as null by using the null element as shown below.

```
<!-- Update a column to null -->
<currentRow>
  <columnValue>4</columnValue>
  <columnValue>Paul Wiew Update Join</columnValue>
  <columnValue>2</columnValue>
  <updateRow>null</updateRow>
</currentRow>
```

Simple class diagram for final design:
High-level system interactions:

Client side

The client side portion of the project went through very few major design changes. The changes that did occur were primarily in how the parameters needed by the server side were sent. For example, the initial design sent arrays of changed data to the Servlets, but the final design has just a single WebRowSet string sent.

Overview

The amount of work done on the client side ramped up much more with the final version of the design compared to the earlier versions. Since the final design uses WebRowSet XML to drive the system, we replaced the HTML table output in the macro with JavaScript code. The JavaScript gets the data to display by making an initial refresh request, which pulls a full WebRowSet XML string. This string is turned into a DOM object that is manipulated as user changes are made. This DOM is then serialized back to a string when making requests.

The overall flow of events for the client-side setup is as follows:

- (Server Side) Macro reads sql-plugin.vm which has a template of the <script> element to put in the page
- (Server Side) Macro generates all parameters and fills in the template,
- The generated <script> has a call to jQuery.sqlPlugin.createTable, which does all the initial setup of object prototypes for each instance of the macro on a page.
  - This script is evaluated whenever the client browser evaluates script tags, meaning that once the script is evaluated, the plugin's objects are instantiated.
  - The object construction process does not immediately send a refresh. It waits for the full document to be loaded -- this is done asynchronously so there will probably not be data, or even a table displayed after createTable() returns.
- Once refresh request has returned, the WebRowSet is put in a DOM object, and a list of objects containing column information is created
- The table data is extracted from the WebRowSet DOM
- The table data, plus column objects are used to construct a "Datatable"
- If the user has editing permissions, the column objects are used to associate various types of editor objects to cells of each column. Additional buttons are added in and around the table to support saving, canceling, etc. At this point the table is ready to be edited.

As the user performs operations, changes are stored in the WebRowSet DOM as described in the examples above. When submitting these changes, the plugin has to take only certain chunks of the changes at a time. For example inserts go alone, without any other update or delete changes. The request is sent after the dialog for inserts has been validated. Any updates that were made, but not saved before an insert remain in the DOM, but are not sent. This is done by making a clone of the DOM, then using jQuery to remove all but the desired sections.

After any kind of change (insert, update, delete) has been successfully saved, the section of the DOM that was sent is converted to the standard WebRowSet data style. For example:
Would be changed to:

when the Servlet returns a success message. The only difference with a delete is that the old data goes away entirely.

Columns and Validators

One issue we ran into was that WebRowSet isn't friendly with error messages. Most of the time it just throws an exception saying there was a 'conflict'. To deal with errors that the user will be able to understand, we had to extract some validation into the client-side. One common error is typing invalid data into a cell. For example a NUMBER cell expects only numbers and other characters for showing numbers. Part of the roll of the column objects that are created after a refresh request is sent is to validate the inputs of the user. So far we have come up with three types of validators that exist: date, numeric, and textual. There could probably be more for other data types, however those three were the most important.

The validators inspect the content of inputs and, if the input is invalid, highlight the cell and display an error message with far more information about what is wrong than what WebRowSet would have given.

Another roll for the column objects is data conversion. Once an input has been validated, it needs to be stored in the WebRowSet DOM. There are some specific requirements for how this has to work, so that values are saved as expected. For dates, they have to be converted into the Unix timestamp format in milliseconds. For varchars, XML entities and tags have to be escaped. This escaping is particularly tricky because when editing, the user needs to be able to see and type the entities directly, but when displaying and saving the non editing version these have to be escaped. For example, the user could embed a <script> element in a cell. If this was not escaped when it was displayed, the JavaScript inside that element could be called, and the page could be broken.

Error handling

When we get errors from JDBC, WebRowSet or even from our own validators, we display them in a small in-page dialog. This dialog generally has a short message describing whatever happened, and depending on the type of error can be clicked for more information. When errors come back from the servlet they are in a small XML format with the exception that was generated and the stack trace, which can be useful for developers during debugging. The error dialog can be closed, so that once the error has been resolved, the user can get rid of the message.

Libraries, usage

To reduce the risks involved in managing all these tasks (sending requests, building and manipulating DOMs, constructing sortable, dynamically changing tables, editing cells, etc) we made use of various JavaScript libraries including: jQuery, jQuery UI, Datatables for jQuery, and jEditable.

jQuery is bundled with confluence. It provides a vast amount of cross-browser compatible operations as well as some more advanced DOM query/manipulation abilities. It is used throughout the browser for manipulation of the WebRowSet and the plugin's HTML contents.

jQuery UI is not bundled with confluence, so a copy was included in our project. It has the ability to create various types of widgets. We used it to create dialogs for insert and error reporting.
Datatables for jQuery is a plugin that allows us to turn an array of data (in our case, the query results) into an HTML table. It supports sorting, adding additional rows, and easily integrated with our code. It also gave examples using jEditable, which provided functionality we wanted but couldn't figure out how to do ourselves.

jEditable lets you build a page that has content that displays as normal HTML content, but when clicked on turns into a form (either preset or custom type) that can then simply change the page, send POST requests, or call an event handler function. We choose the event version, to convert data for WebRowSet, and built several custom input types to handle our specific validation needs.

Testing

Testing the JavaScript can be tedious. Since so much of it is event based, you have to simulate various events occurring in the tests, almost like writing a script for how someone might use the page. There are some cases where simple unit testing was easy. For example some of the DOM operations and data conversions were easily tested using qunit.

Another issue with JavaScript testing is that all data has to be mocked because you cannot make JavaScript requests to servers other than where the page originated. This means we would either have to install the qunit tests to the server each time we ran them (not nearly as easy as it sounds, it turns out), or view the file locally and mock out the requests to look at raw XML files. We choose the latter as it was much easier to get working in the end, even though it can be tedious to set up all the files/data needed for new tests.

Process and Product Metrics

Velocity - This is a measurement of how fast work is being completed. It was determined by the amount of story points completed in the previous sprint. This was used during the planning stages of future sprints to determine how many user stories will be targeted for that sprint.

User Story Status/Impediments - We recorded user stories and their associated status and current impediments in a table. This was used in order to keep track of associated risks, progress, overall impediments, and associate task ownership for a given user story.

Test Pass/Fail - Test related information was recorded in order to track project health. In our methodology user stories cannot be considered complete until there are passing tests in place for the functionality in that user story. Code coverage information was generated by running unit tests.

Time/Activity Tracking - Time/Activity tracking was used to determine day to day activities and sprint process. Aside from being part of the RIT senior project requirements this measurement contributed to the daily burn-up chart (described below).

Burn-up Chart - This chart mapped total spent time relative to the total estimated time for a sprint. This chart was updated daily. This measurement also helped make sure estimates were on track while validating estimations for a sprint.

Burn-down Chart - This chart mapped remaining story points to be implemented in the sprint against the estimated total. This gave a visualization for how much remained to be completed for the sprint as well as determine if goals were still on target. It was updated at daily scrum meetings by the scrum master.

Product State at Time of Delivery

All required user stories were completed for this project. We are delivering a product that Xerox requested and we have even had the chance to complete some user stories that were mentioned as "nice to have" if time permits during the initial requirements workshop. This includes a Confluence Wiki macro that allows the user to perform insert, update, and delete operations on tables containing results from a database query. We were also able to handle null values, basic security, and a fancy format that allows paging and searching, allowing larger tables to be handled in a more manageable fashion. We were unable to complete one "extra" which included tying the Confluence users accounts to their database user accounts.

The plugin now resides within the Confluence plugin repository and is available to Xerox and also any user of Confluence. This means that the project is now open source and can be maintained through the Confluence plugin repository. There were no discrepancies between what was delivered and what was promised and our product backlog has been completed. Having completed the prioritized list of user accounts...
stories on the product backlog lets us know that we have completed everything that Xerox has requested since the ownership of the product backlog artifact is our Xerox sponsors.

**Project Reflection**

This project went very well overall. We had regular post-mortems at the completion of each sprint and the pros outweighed the cons a majority of the time. The methodology chosen was executed well and resulted in productivity and clear goals for day-to-day implementation. The team also worked well together and the rotation of pair programming teams helped in maintaining consistency through sprints but also added variety within the project.

There were issues in the project but were typical of any development effort. Most of our issues were a result of having to develop the plugin for compatibility in both Firefox and Internet Explorer and also the fact that we began project development on MySQL and that did not translate well to Oracle. We would develop and test on Firefox and occasionally check on Internet Explorer to find that many components were broken. Also, we spent an entire sprint in translating current functionality from MySQL to Oracle, something that could have been avoided by developing with Oracle the entire time. Some other issues came in the form of poor documentation on the use of WebRowSet and also an initial plugin design by Bob Swift that did not lend itself to our project. In addition to documentation concerns with WebRowSet, there were flaws in Sun's WebRowSet implementation as well as unexpected behavior of Oracle and MySQL JDBC drivers.

Looking back at the issues that were experienced there are a few specific things that would have been done differently. First we would have moved to the Atlassian community earlier. This would have allowed us to provide a more full history of our product development. We could track bugs through Atlassian and also make it easier for Xerox to install and test our solution throughout the development period. Second we would have had more thorough technology spikes where a better breadth of testing would take place. In our initial design change (as the result of technology limitations) we were only looking for a solution to our current problem, a more vertical search for needed functionality. In our second design change we were more wary in trying to make sure this would be a final solution. In this case we ensured that future user stories would be possible with the technologies that were being chosen (meaning we did look-ahead testing for insert and delete even though at the time update functionality is what we needed). Lastly, we would have implemented client-side testing with QUnit earlier. This testing, primarily for JavaScript, would have allowed us to improve testing speed and also make regression testing easier. These QUnit tests would have also assisted in speedily checking browser compatibility for our product (in some respects).

Overall there were many things that went exceptionally well within our project. These included our process choice, communication, coach and sponsors, technology choices, requirements gathering, and our move to the Atlassian plugin repository. Our process choice was simple yet effective and provided us with good direction and risk handling for our project implementation. Communication went smoothly throughout the duration of the project. This subject is hit upon more directly within the following section "Collaboration Techniques" but our combination of in person sprint meetings and email with our sponsor, coach and peers allowed us to operate at a high level of efficiency. The coaching and suggestions from our sponsors and faculty advisor were typically brief but to the point and provided concise direction and solutions for team issues and strategies. Our initial requirements workshop was effective in identifying nearly all of the existing user stories now in the product backlog. There were no real surprises during the course of the project and we always knew what was coming next and where we stood as far as project progress. Finally, our move to the Atlassian plugin repository went very smoothly. We had anticipated some problems but it turned out to be easier than we thought and we are now an open source project available to Confluence users. We can build and deploy our project to Confluence and we have moved our bug tracking to that space.

**Collaboration Techniques**

- **Wiki** - The ability to use the Confluence Wiki as well as develop the plugin for it gave us a slight advantage. We became familiar with how it worked and what we were developing for and were also able to have an easily accessible area for project artifacts. This was the main location for our methodology and project artifacts and helped keep the team in sync. We were able to allow our sponsors to have access to this space in addition to the team so visibility of progress and where we stood at all times was reflected through the wiki.

- **E-Mail** - E-mail was critical for the periods of time in between sprint meetings. We found it useful since we could typically refocus efforts elsewhere or write e-mails overnight in order to have a response by the next day. We had few time critical issues and e-mail worked well in allowing the
sponsor to respond when they had time (which was typically within a day). This worked well since we were not close enough for constant face to face interaction and calling by phone would not allow us to collect our thoughts and may conflict with others business schedules.

- Adobe Connect - Adobe Connect was used during sprint meetings to share the desktop of a team member to drive the meeting. We had early issues with sound and video, both of which were not critical or we could not get cameras or microphones. We used Adobe Connect for the desktop sharing and used the speaker telephone for the remote participants. It was an easy and useful way to show our progress to remote participants but we did not rely on functionality other than the desktop sharing capabilities.

- Telephone - Telephone was used to allow audio in sprint meetings between the on location and remote meeting participants. Telephone in other interactions was usually as a last resort (calling to see if they would attend the meeting, see where someone was) and was not a primary method of communication or collaboration. This is possibly the result of successful scheduling and reliable team members that telephone was not primarily needed.

- In-Person - All sprint and scrum meetings were times where in-person collaboration was important. Most sprint meetings were only missing Peter Alfvin (due to his location) as far as in-person attendance. The ability to have face-to-face contact and quick responses during development (within team) and scrum meetings (with sponsor) was essential to keeping on track and correctly planning project and sprint time.

References

5. WebRowSet API Documentation