Sample Tracking
Database Plug-in

Database Values Plugin

This is the existing database integration plug in. Its primary purpose is to use a database to populate a select list.

The database values plug-in initially looked like a good starting point for integrating the data in the GLK with JIRA. It allows a field to store a, hidden, key that is used to look-up the value to display when viewing the field. It also allows listing of the display values to select which key to use. For Sample Tracking the key to use is tied to the Sample's issue and shouldn't ever change, more over the user is not expected to select which GLK Sample entry corresponds to the JIRA Sample entry. The compound key used for the 'summary' field contains enough information to infer which GLK entry to use.

Accessing Data in the GLK

The GLK entries are keyed on *Extent_ID* and *Database*. The GLK stores all of the information in the *ExtentAttribute* table in the column called value. This table is keyed on *Extent_id* (to select the correct sample in our case) and *ExtentAttributeType_id*. *ExtentAttributeType_id* is a foreign key from *ExtentAttributeType*. It is not human readable but *ExtentAttributeType* contains a column called *type*, that is accessible via a join, that is human readable. The query for selecting an attribute for a sample is:

```
SELECT Extent_id, value FROM ExtentAttribute
WHERE Extent_id = '<Extent ID>' and ExtentAttributeType_id = IN
      SELECT ExtentAttributeType_id FROM ExtentAttributeType
      WHERE type = '<attribute name>')
```

Extent_id is also returned as the plug requires the query to also return the ID that it will use in future to get the value to display.

Mapping between JIRA and the GLK

DB Query

The fields in JIRA store *Database Name*, *Collection Code* and *BAC ID* but not *Extent_ID*. The *Extent* table can be used to map between *BAC ID* and *Extent_ID*. It contains a property called *ref_id* which, in the case of extents that represent samples, is the *BAC ID*. The query to convert a *BAC ID* to an *Extent ID* is:

```
SELECT Extent_id, Extent.ref_id FROM Extent WHERE ref_id = '<BAC ID>';
```

As *ref_id/BAC ID* is being used to select the record it must be included in the output for the plug-in.

Externally Converted

If created from FLURP or FLURP's output then Extent ID could be inserted into the field directly.

Summary

Positive

- Already Exists
- While not designed for displaying linked values the only difference is setting the link ID in code instead of letting the user choose.
- Comes with a working search
- Uses it's own database connection and so doesn't need any configuration of the server.

Negative

- Assumes only one ID is needed; we need two, one for the db and one for the Extent ID / BAC ID.
- Only some of the paths have access to the Issue object, which could be used to access the extra ID data.
  - The velocity templates; (view/edit/search)_databasevalues.vm all have access to the Issue as the variable $Issue but none of them pass it to the customfield.
  - Searcher and Fallback Searcher will never have access to the Issue. (Therefor any data needed to place the values in order must be present in the key/ID)
  - JQLSearch doesn't have access to an Issue as its not associated with a specific issue. As it can use multiple 'parameters' this is less of an issue.
- Only some of the paths go through a substitution stage which could be used to add the extra ID data into the query.
- The method used to pass the variables for substitution to the query doesn't appear to be thread safe
- Configuration is all via text files which require access to the server to change
- Only single quotes can be used in the SQL double cause it to fail with almost no feedback.

Notes:

For using multiple databases together; Union and using the format <db>::<table>

For multi-selects would exists in and a list ('A','B',...) be more efficient, as it is only a single query?

Rows vs Columns

When the plug-in was only going to fetch the data one value at a time it didn't matter where the rest of the data for that issue was stored. The data could be stored as values in columns within the same row or as name value pairs.

Row based data (1 cell)

```
SELECT id, value FROM #test WHERE name='X' AND id='1';
```

Column based data (1 cell)

```
SELECT id, X FROM #test WHERE id='1';
```

While the SQL to generate the results varies between the two they both return exactly the same data:

id	value
1	A

As the data returned to the program is identical for the two arrangements the program only needs one routine to handle this data.

Only one value was needed when each field would be carrying out separate queries. Actually two values were needed id and value but in both schemes the id field is available in the same row as the value.

However, when the number of fields increases it is no-longer possible to express the data identically, without first translating it. Naively adding to the queries for 1 cell results in:

Row based data (2 cell)

```
SELECT id, name, value FROM #test WHERE name IN ('X','Y') AND id='1';
```

id	name	value
1	X	A
1	Y	B

Column based data (2 cell)

```
SELECT id, X, Y FROM #test WHERE id='1';
```

id	X	Y
1	A	B

The code to process the values now needs separate versions for the column and row formats.

- The row based data returns two rows and the code must inspect the added name column to tell which field the data in value is for.
- The column based data has each fields value read from a separate column.

Initially only the GLK will be used and the GLK is row based. Outside of the GLK; column based tables form the majority databases. If the plug-in was to be reused or published it would be expected to handle column based tables. Publishing would be with the intent of sharing testing and maintenance costs.

Columns to Rows

Before

Id	X	Y	Z	timestamp
1	A	B	C	0x00000001009112d8
2	a	b	c	0x00000001009112d9

SQL

```
WITH col_table (id, X, Y, Z, timestamp)
AS
(
    SELECT id, X, Y, Z, timestamp
    FROM #test
    WHERE id='1'
)
select id, 'X' AS type, X AS value, timestamp from col_table
union all
select id, 'Y' AS type, Y AS value, timestamp from col_table
union all
select id, 'Z' AS type, Z AS value, timestamp from col_table
;
```

After

id	name	value	timestamp
1	X	A	0x00000001009112d8
1	Y	B	0x00000001009112d8
1	Z	C	0x00000001009112d8

Notes:

'WITH' is also known as a Common Table Expression (CTE). It allows a shared restriction, such as id='1' to be evaluated only once. Without it each select would require a WHERE id='1'. col_table in this example should only be one row, making the select statements trivial.

- A SELECT is required for each column that is transposed into a row.
- Only the col_table select is carried out on the whole table.

converting rows into columns

Before

id	name	value	timestamp
1	X	A	0x00000001009112e0
1	Y	B	0x00000001009112e1
1	Z	C	0x00000001009112e2

SQL

```
SELECT id,
      MAX(CASE WHEN name='X' THEN value END) as X,
      MAX(CASE WHEN name='Y' THEN value END) as Y,
      MAX(CASE WHEN name='Z' THEN value END) as Z,
      MAX(timestamp)
FROM #test
GROUP BY id
WHERE id='1' AND name IN ('X','Y','Z');
```

After

id	X	Y	Z	timestamp
1	A	B	C	0x00000001009112e2

Notes

Conversion from rows to columns also requires a long piece of SQL, with a MAX and CASE clause per output column. The blocks within the MAX statement are evaluated once per row that contains the specified id. The IN clause restricts these rows to only ones that have the properties required. In the example each CASE statement and the timestamp are evaluated three times. When the CASE statement fails to match it evaluates to NULL. Each of the CASE statements should match once, and only once. The MAX then takes the values from each of the runs and picks the highest, with NULL being the lowest possible value the value from when the CASE matched will be used.

Table conversion conclusions

The SQL required to convert the 'shape' of a table is very verbose and quite complex. The hope was that there would be a simple way, involving only modifications to the configuration, to make the plugin work with both row based and column based data.

- Conversion of the data at the SQL level is more complex and less efficient than conversion in code.
- Conversion in code requires two different code paths.
- Only row based data needs to work initially.

Timestamps

The actual name varies between the database vendors but for SyBase a time-stamp field is one that is automatically updated with a number (that is larger than any number previously used) each time an insert or update occurs. The original purpose was to simplify the creation of optimistic locking schemes. It is also useful however for our situation where we only want to update the minimum number of index entries, to minimize the 'hit' of synchronizing and so to make more frequent synchronization practical.

Sybase

Sybase implements the time-stamp field as an 8 byte value stored in a special column, which is always called 'timestamp'?

Examples

```
CREATE TABLE #[1]test(name CHAR(10), value CHAR(10));
ALTER TABLE #test ADD timestamp;
INSERT INTO #test (name,value) VALUES ('1','one');
SELECT * FROM #test ;
```

name	value	timestamp
1	one	0x00000001007effb8

```
UPDATE #test SET value='1' WHERE name='1';
SELECT * FROM #test ;
```

name	value	timestamp
1	1	0x00000001007f005d

```
SELECT * FROM #test WHERE timestamp > 0;
```

name	value	timestamp
1	1	0x00000001007f005d

```
SELECT * FROM #test WHERE timestamp > 0x00000001007f005d;
```

name	value	timestamp

```
INSERT INTO #test (name,value) VALUES ('2','two');
SELECT * FROM #test WHERE timestamp > 0x00000001007f005d;
```

name	value	timestamp
2	two	0x00000001007f0331

```
UPDATE #test SET value='one' WHERE name='1';
SELECT * FROM #test WHERE timestamp > 0x00000001007f005d;
```

name	value	timestamp
1	one	0x0000000100800371

1 Table names beginning with '#' indicate that the table should be removed when the session finishes.

2	two	0x00000001007f0331

Conversion

SyBase stores the timestamp as 8 bytes (64bit). It can be converted to BIGINT, which is converted to Long in the JDBC drivers. The conversion is done with:

```
CONVERT(BIGINT,timestamp)
```

```
CONVERT(VARBINARY, ?)
```

The Long values created don't maintain the ordering of the updates but they do convert back to the right number. This could cause problems though when attempting to get the max timestamp value for the next query. Applying the max function to the BIG INT values wouldn't necessarily return the value that mapped to the maximum hex value.

	Hex value	BIG INT	Converted back
older	0x00000001008195a5	-6515159448904794112	0x00000001008195a5
newer	0x00000001008195a8	-6298986666791010304	0x00000001008195a8

Alternatively using Numeric; the conversions from and to timestamp are:

```
CONVERT( NUMERIC(20), timestamp )
```

```
CONVERT(VARBINARY(8), CONVERT( NUMERIC(20), ? ))
```

The values from numeric are ordered correctly. The value for timestamp is a 64bit unsigned number and so it could be too large to fit in a Long. Therefor it would be better to use BigNumber when retrieving it via JDBC.

	Hex value	NUMERIC(20)	Converted back
older	0x00000001008195a5	282031538110464	0x00000001008195a5
newer	0x00000001008195a8	282031538307072	0x00000001008195a8

- byte[]: Native Very SyBase specific
- Long: Via convert(BIGINT) lacks ordering
- BigNumber: Via convert(NUMERIC(20)) The best option.
- Date/Timestamp: No

MySQL 5+ and Timestamps

MySQL 5+ will create a column whose value is automatically updated if any other value in that row is changed.

```
CREATE TABLE t (ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                             ON UPDATE CURRENT_TIMESTAMP);
```

MySQL provides access to the value using the same format and functions as a 'DATATIME' column. By default the format in SQL is 'YYYY-MM-DD HH:MM:SS'.

Conversion

Conversion to a number can be done with:

'UNIX_TIMESTAMP' and 'FROM_UNIX_TIMESTAMP'

- Long/BIGDecimal: Converted Closest to the SyBase values
- Date/Timestamp: Native A date is clearer for debugging than a twenty digit number

Oracle and Timestamps

As is to be expected Oracle don't support standard `TIMESTAMP` columns. They have a data type called `TIMESTAMP` that will allow the table to be created but it is not automatically set on insert or update. The closest equivalent is the `ORA_ROWSCN` pseudo column. It provides the highest SCN (System change number) that is associated with a change to the row. Actually by default the SCNs are associated with storage blocks and so a rows SCN would be changed is any row that shares a storage block with it has changed.

The SCN granularity is set on the table by declaring it with '`ROWDEPENDENCIES'` or altering it to have '`ROWDEPENDENCIES`'.

```
CREATE/ALTER TABLE t (
     ...
) ROWDEPENDENCIES;
```

Behavior

SCN	Without ROWDEPENDENCIES	With ROWDEPENDENCIES
new = old	The row is guaranteed not to have changed	
new > old	The row may or may not have changed	The row is guaranteed to have had an insert or update occur. (The update may not have changed anything though).

Note: One SCN may be used for multiple rows, it cannot be used as a unique id.

Conversion

The SCN number can be converted to a date using the function `SCN_TO_TIMESTAMP(ora_rowscn)`. This uses an internal look-up table. Oracle stores about 5 days worth of information, if the look-up has been dropped from the table `SCN_TO_TIMESTAMP` returns an error. It is therefor safer to use the SCN number directly. It also isn't if it is possible to convert back again for the next select.

- Long/BigDecimal: Native — Simplest
- Date/Timestamp: Using SCN_TO_TIMESTAMP — Danger if no updates occur for 5 days

  Conversion back may not be possible

DB2

From version 9.5 an auto-updating TIMESTAMP column type is available.

```
ALTER TABLE myschem.mytable
      ADD update_ts TIMESTAMP NOT NULL
      GENERATED BY DEFAULT FOR EACH ROW ON UPDATE AS ROW CHANGE
      TIMESTAMP;
```

The DB2 datatype stores the time to the nearest microsecond which should be accurate enough to avoid two updates to the same row getting the same time stamp, although it's not as good as an auto incrementing value. The value is in the format:

`YYYY-MM-DD-hh.mm.ss(.z{1,12})?`

Conversion

To convert it into a number requires a few functions, and converting it back again is even more complex.

The JDBC driver only excepts values of type java.sql.Timestamp.

- Long/BigDecimal: Conversion via stored procedures Complex conversion each way
- Timestamp: Native All but DB2 would use BigDecimal

Conclusion

With the exception of DB2 all of the databases have features that can be used to produce a column that is automatically updated for any row that is modified via insertion or update.

Oracle adds a further caveat, an update to the timestamp does not necessarily mean that anything in the row has changed. For updating the Jira / Lucene index this won't cause errors, although it could create performance problems. Initially it will be assumed that it is rare for a row to have a modified timestamp and not actually need its index updating. This can be revisited later once the system is in use and statistics can be generated to test this hypothersis.

SQL Variable Replacement: Field Names

Each field needs to add to the central SQL query a section to retrieve its value, along with the others. The challenge is making this mechanism both simple and functional. Examples of situations that could arise:

- Column based tables need the fields to be listed as col1, col2, col3 where as row based tables would need them to be wrapped in quotes 'col1','col2','col3'
- The list of columns doesn't have a separator at the end. If the separators are output by the fields then one of the fields needs to know that it is last, or one know that it is first (when prepending).
- Some fields may not be a single value, but instead could be the result of merging other fields values: col1 || ' ' || col2
- A field may require a join or change to the main select

Velocity SQL Templates

The velocity engine is tied into JIRA and so is guaranteed to be available. It provides methods for looping and can handle nested parsing. Example of possible object structure:

AllFields	a map from field names to Fields
DBFields	a list of DBFields
Field	with properties: name, value
DBField	extending Field with properties: template

Where template was a string containing the template for each field.

An example Template

Velocity can be used at its simplest for search and replace, using values placed into the context.

Velocity Template

```
SELECT id, name, value FROM ${allFields['Database']}::#test
WHERE id IN (
      SELECT id FROM ${allFields['Database']}::#test
      WHERE name='JIRAid')
```

Result

Assuming that the field 'Database' had the value 'giv3':

```
SELECT id, name, value FROM giv3::#test
WHERE id IN (
      SELECT id FROM giv3::#test
      WHERE name='JIRAid')
```

A more complex Template

Velocity has looping and conditional statements that allow for a more dynamic SQL statement which vary based on multiple factors.

Velocity Template

```
SELECT id, name, value FROM #test
WHERE name IN
#set( $first = true)
( #foreach( $field in $fields )
      #if ($first)
            set($first = false)
      #else
            ,
      #end
            '#parse($field.template)'
#end )
```

Result

Assuming that the templates for X,Y and Z only contained the name of the parameter the above would parse to:

```
SELECT id, name, value FROM #test
WHERE name IN
('X','Y','Z')
```

Notes:

- The Velocity templates are simple when only search and replace is needed but can quickly get complex if other functions are used.

SQL

Single Fields Value for a Single Issue

It doesn't make sense with the central update service to directly update a field. This reflects work done before the decision to use polling.

```
SELECT value from ${DB}..ExtentAttribute JOIN ${DB}..ExtentAttributeType ON
ExtentAttributeType.ExtentAttributeType_id = ExtentAttribute.ExtentAttributeType_id
WHERE type = 'fieldname' and Extent_id = ${Extent_id};
```

The join in the above statement could be removed by caching the value of ExtentAttributeType_id (for that DB)

```
SELECT ExtentAttributeType_id FROM ${DB}::ExtentAttributeType WHERE type =
'fieldname'
```

Initializing the fields

Typically when an issue is created none of its database fields have values, (later versions may provide fields and selectors for the issue creation and editing pages). The key considerations for initializing the fields are:

- Who is responsible for setting the values
- How are the values generated
- When and how is the code to set the values triggered

Responsibility for Initial Values

There are three 'units' that could carry out the actual setting of the fields' values:

- The issue creator (Human or Program)
- A dedicated initialization unit
- The synchronization unit

The issue creator (Human or Program)

The program creating the issue may already know the fields' values and could pass these into the issue as part of its creation.

- No further database permissions are needed by the plugin, as it doesn't directly access the database.
- The timestamp field's use remains consistent
- No need to identify the database rows that the data came from
- The logic, for creating the fields' initial values, is not the same as the logic used to update the field.
- The logic is stored separately to the logic in the plug-in.
- A Human entering the values could introduce errors and inconsistencies

A dedicated initialization unit

A separate piece of code/logic could be used to lookup the initial values. This would involve using two separate queries.

- Each unit only does one thing
- Only the SELECT permission is needed by the code
- The timestamp field's use remains consistent

- The fields are only programatically set, removing human error.
- The logic, for creating the fields' initial values, is not the same as the logic used to update the field.

The synchronization unit

The timestamps on the fields for the new issue are unlikely to be new enough that the synchronizer would update then fields of the issue. The timestamp could however be modified to ensure that the values were 'updated' to their initial values. The timestamp can be updated without changing the values by assigning a value to itself. (e.g. UPDATE #test SET value=value WHERE name='1')

- The same logic, in the synchronizer would be used both for the creation and update of a field's value.
- A separate unit would still be required to modify the timestamps.
- Inconsistent use of timestamp. Normally timestamp is considered co-variant with the row's contents. e.g. timestamp != old_timestamp only if the row has changed. This would result in timestamp != old_timestamp but with no change to the row. This change wouldn't be a problem for the DB plug-in but could 'surprise' other systems using the field.
- The unit that modified the timestamps would require the UPDATE permission. Unlike the currently required permissions, INSERT (for adding the JIRA ID on issue creation) and SELECT (for the synchronizer), UPDATE is destructive. INSERT can only add extra data; its modifications to the DB can be removed, if needed, to return to the state the database was previously in. UPDATE, however, removes data. The previous field values are gone forever once an UPDATE has taken place. It is not possible to unpick changes made by UPDATEs.

Generation of Initial Values

Each of the units that are described in 'Responsibility for Initial Values' would employ a different method to generate the value for the field.

The issue creator (Human or Program)

If a human is generating the values then the actual method of selection may be very complex. The method with which they enter the system would be via the Issue creation form page. An input of some form would be needed for each field. Ideally the input would provide a closed list of options, but boxes for direct entry of values would also be needed.

For a program inserting the values the JIRA CLI could be used with the extra fields specified in the CSV file. The values inserted could have come from the database directly or have been processed in some way. Ideally the program would pass the raw database values and the logic for deriving the displayed values could be shared with the Synchronizer.

A dedicated initialization unit

The logic to find the values would have to include a method of determining the DB row ID that matched the current issue (The opposite of the synchronizer looking up the Issue ID from the DB ID). This would probably be expressed as an SQL statement. The formatting of the values returned by the SQL Statement could be done with the Synchronizer (for consistency). It would also be possible for the values to be modified and formatted within the SQL, although it would probably lead to differences between the initial value's format and the format of any update.

The synchronization unit

The synchronization unit already has to format the results of it's queries for modified values. The values returned from the query could have been modified by the query (although this could introduce inconsistencies if a separate initialization query was used). The actual values returned could then be formated, the range and methods of formatting possible with Synchronizer are explored later.

Triggering of the Generation of Initial Values

Create Issue Page

Seemingly the obvious place, but the Create page is only actually displayed for human created Issues

View Issue Page

- Used for both JIRA created and externally created
- Issues created outside of JIRA may wait a long time before being viewed
- Can't be searched for until it has been viewed.

workflow

A function can be triggered automatically when an Issue is created. A function that is automatically called when a state is entered is defined:

The create function can't have actions added and so the function is added to the first step. As this should only be called once the first step is 'hidden' with the Issue only remaining at that step long enough to run the function and then automatically moving to the step that a user would consider the first step. Auto move:

- This would ensure that the fields of issues created within JIRA used the same logic as those created by VAPOR.
- Issues will need to be assigned a modified workflow to use this function

Other

Is there an action or whatever is used with the rest interface that I could intercept to avoid having to edit lots of workflows?

Configuration

The fields need to be able to share a common set of db connection information.

JNDI would be best for this, is it possible to add to the JNDI environment from our plug-in?

Ideally it would use prepared statements to avoid SQL injection. With one field mapping to one parameter of the statement.

The challenge is that tables can't be parameters (I think!) which makes the use of DB:: harder.

Some of the fields may need preprocessing before being used in the SQL.

Field -> Velocity -> prepared statement parameter

Field -> Velocity/Lookup table -> select DB connection

Alternatively making it very specific to getting GLK fields

A dropdown with the ExtentAttributeTypes and just select one of them. Wire it with DB -> table name or connection and BacId -> param.

- limited functionality

+ simple to configure

Notes

Timestamp needs to be stored somewhere. Using it hardcodes the assumption that it will be used with polling.

Adding a Filter to trigger re-indexing

Look at the workflow function in jira that syncs the index at the end of a transition