Schoolzilla Homework Assignment

Assumptions

- Input format of data will be in CSV format with the first row reserved for column headers
- Exactly one column always contains id values that can be used to uniquely identify the “owner” of an input row (Foreign Key)

Solution

Challenges

- Need to have a repeatable way of defining how source tables are to be transformed. The mapping definitions need to be importable/exportable so that they can be re-used for additional tables in the same format.
- UI design must support defining the table mappings in a relatively simple and intuitive way.
- Data must be verified according to type during the transformation process. Any rows with invalid data need to be flagged and aggregated into some sort of error/kickout report so that users can correct any problems and reprocess.
- Design must support parallelizing processing of large datasets to allow for easy scaling.
- Design needs to allow for the possibility for batch style processing of particularly large datasets. I don't necessarily see this as an immediate need, but the design should be flexible enough to allow for it.

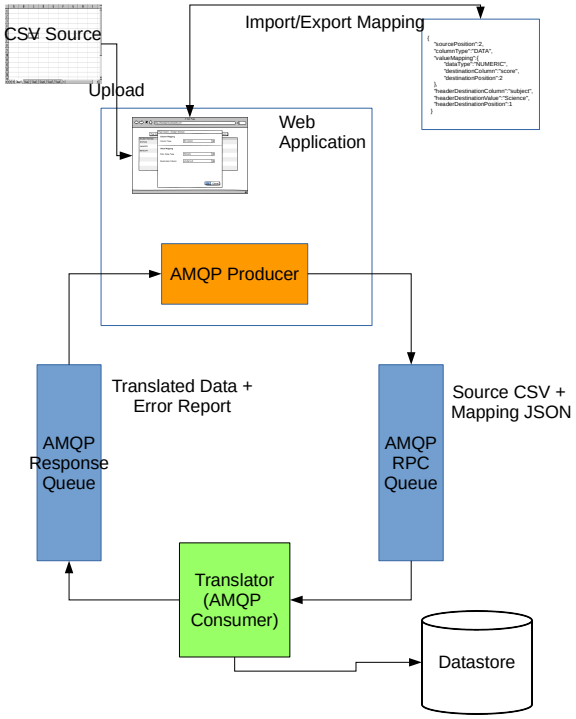
Tools/Libraries/Etc.

- CSV parsing library – I used opencsv (http://opencsv.sourceforge.net/) but any library providing similar functionality should be fine.
- JSON/Object mapping library – I used GSON (https://code.google.com/p/google-gson/) in my PoC but Jackson or anything similar would be an excellent choice as well.
- Messaging Framework (AMQP) – I personally am partial to RabbitMQ (http://www.rabbitmq.com/) as I've had experience with it in the past but ActiveMQ or any other AMQP provider would also be fine.
- Javascript frameworks – In order to accomplish a lot of the more basic functionality for the web UI we can leverage javascript frameworks like jQuery and jQuery UI to do a lot of the heavy lifting. A couple of plugins in particular worth calling out are SlickGrid (https://github.com/mleibman/SlickGrid/wiki) and jqGrid (http://www.trirand.com/blog/) both of which provide an impressive range of functionality for displaying, sorting, and editing data in a grid.

Solution Overview

The customer entry point for our solution will be a web application that provides functionality to upload, view, and edit source data in the form of CSV files with headers. Once data has been uploaded, the user will be able to define column mappings for transforming the data into the appropriate destination table format. Once the mappings have been defined, they are packaged up with the CSV data into an AMQP message which is then placed on a queue. The message will then be

picked up by the Translator (AMQP consumer) responsible for transforming the source data according to the included column mappings. Once the Translator has processed and transformed the data the output will be written to the datastore and the output (including any errors) will be placed on a response queue for the web application to consume. The web app can then display the output data and an error report if necessary.



Solution Detail

JSON Column Mapping

The heart of my solution revolves around JSON based meta-data used to define how columns from the source table are transformed and mapped to the destination table. The meta-data also defines validations that should be performed on the column values during transformation.

Sample JSON Column Mapping

```
{
   "sourcePosition":2,
   "columnType":"DATA",
   "valueMapping":{
        "dataType":"NUMERIC",
        "destinationColumn":"score",
        "destinationPosition":2
   },
   "headerDestinationColumn":"subject",
   "headerDestinationValue":"Science",
   "headerDestinationPosition":1
 }
```

Attribute Name	Description
sourcePosition	Position (index from 0) of the source column the mapping applies to
columnType	Type of column being mapped. Valid values are “ID” and “DATA”
headerDestinationColumn	Destination column that the header value will be mapped to
HeaderDestinationPosition	Position (index from 0) of the destination column the header value will be mapped to
headerDestinationValue	Value to be inserted in headerDestinationColumn
valueMapping	Sub-document that defines the mapping for the data values of the column
dataType	Type of data contained in the source column. This is used for validation of the data during transformation. Valid values are “NUMERIC”, “STRING”, and “BOOLEAN” (though this is easily expanded to include additional data types)
destinationColumn	Name of the destination column the source values will be mapped to.
destinationPosition	Position (index from 0) of the destination column the source values will be mapped to.

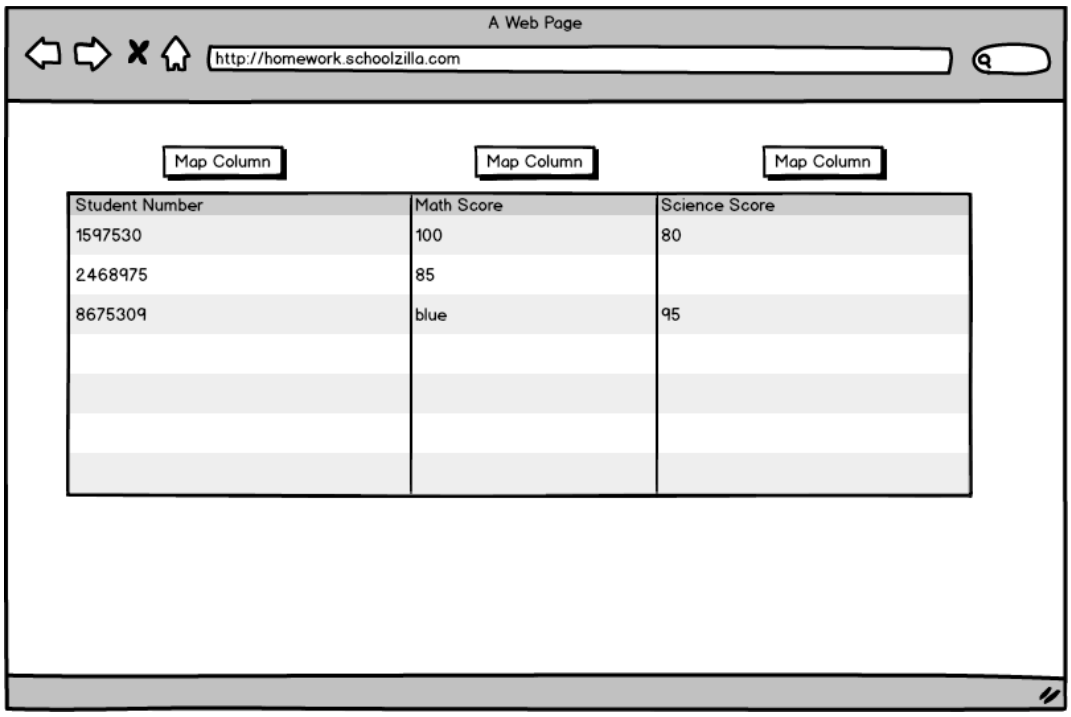
Web UI

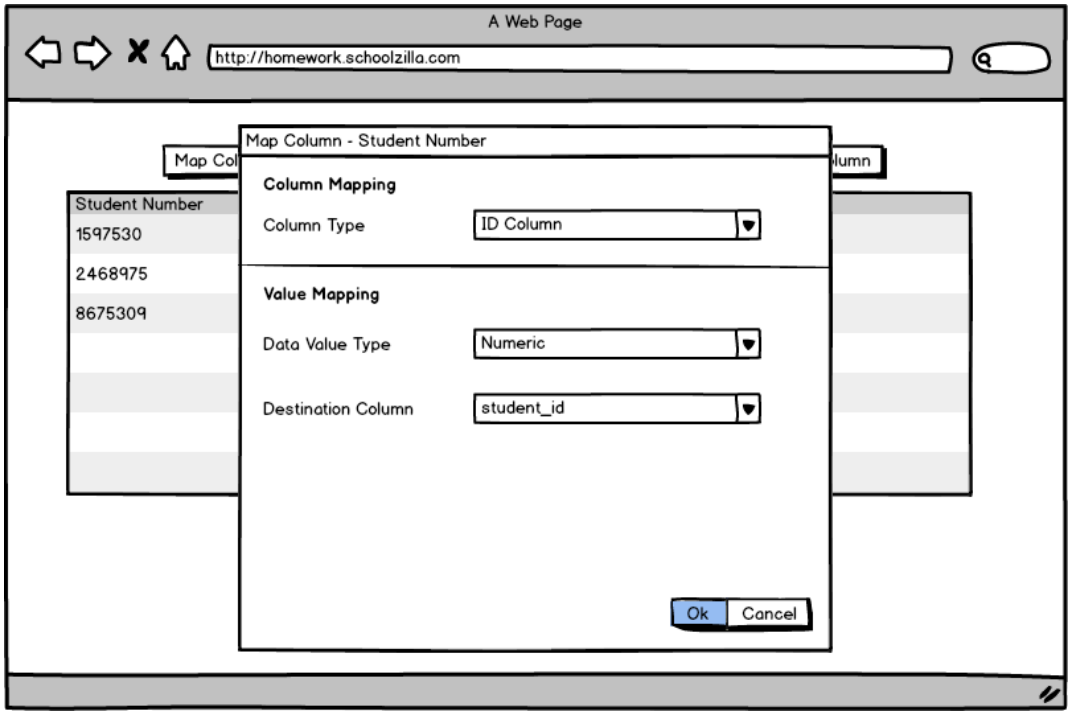
The web UI will need to provide users with the following functionality:

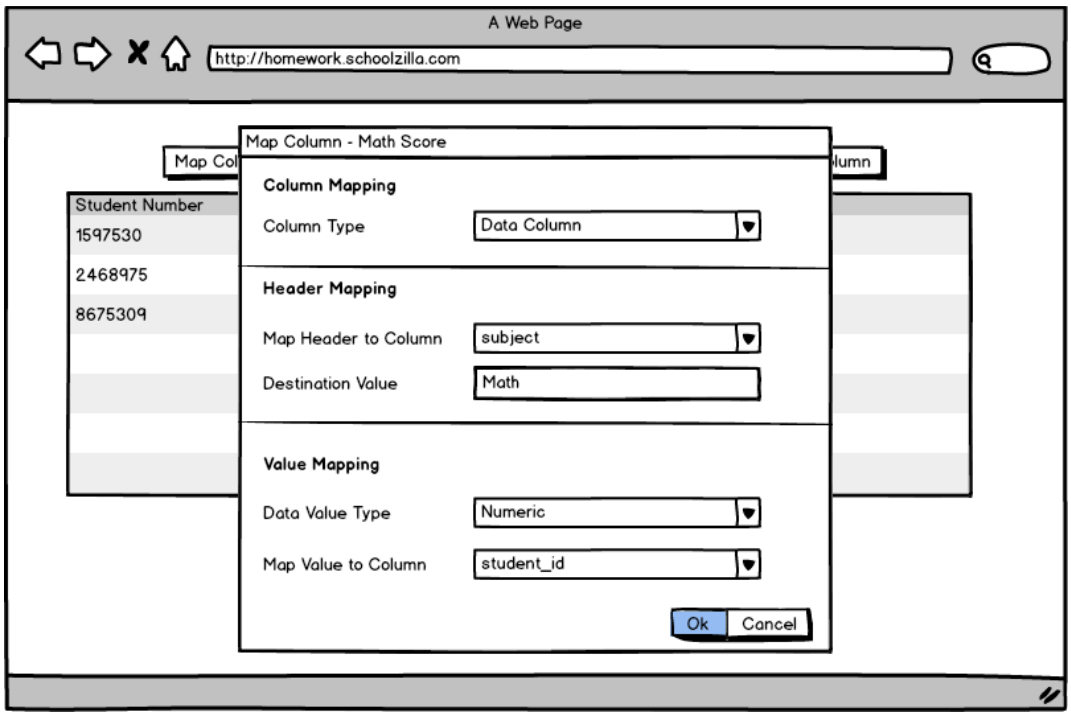
- File upload of the CSV source data
- Display of CSV source data in an editable grid
- Ability to select destination table for data transformation
- UI for defining column mappings for the data transformation
- Export of JSON column mappings
- Display of transformed data and error reports

Most of this functionality (file upload, display of data in a grid, etc.) is available through various javascript frameworks such as jQuery and jQueryUI so I don't want to spend too much time focusing on it. To me, the key aspect of the UI design is how to provide users with a straightforward way to configure the column mapping definitions. This can be handled with a UI that walks the user through defining a mapping for each column on the source table which will then be converted into the appropriate JSON format. The resulting column mapping definition can also be exported so that it can be used to run later transformations or for batch processing of large amounts of data.

I have included a few wireframe screen mockups to provide a general idea of what such a UI might look like:


A Web Page
http://homework.schoolzilla.com
Map Column
Map Column
Map Column
Student Number
Math Score
Science Score
1597530
100
80
2468975
85
8675309
blue
95


A Web Page
http://homework.schoolzilla.com
Map Column - Student Number
Column Mapping
Column Type
ID Column
Value Mapping
Data Value Type
Numeric
Destination Column
student_id
Ok
Cancel
Student Number
1597530
2468975
8675309



After selecting the destination table that data is being mapped to, each column will have a button to initiate mapping. Id columns and data columns have a slightly different set of fields that need to filled out, corresponding to the information that needs to be included in our JSON column mapping fields. Some attributes, such as column positions, are filled out behind the scenes to ensure accuracy and to simplify the UI as much as possible.

Messaging (AMQP)

In the context of the web application, we want to return results back to the UI for display to the user in addition to saving them off into the datastore. To facilitate this we'll need to setup AMQP using an RPC style request/response configuration. Messages containing source data and column mapping definitions are placed on a work queue to be consumed by the translator for processing. Once processing is complete, a response message containing output data and error information is placed onto a separate response queue. The response queue is tied to a specific request, and the application will block while waiting for the response to come back. We sacrifice some potential speed using this approach since the client must wait for all processing to complete before receiving a response. However we still have the ability to easily scale to handle multiple requests simply by adding additional translator consumers. Provided the datasets being processed are reasonable in size there should be no noticeable performance hit.

Translator

The translator component is responsible for taking the column mapping JSON and applying the transformations to the rows from the source table. Additionally the translator performs validation for both the value columns (enforcing data types) and for ids (validating against an external source).

For additional implementation details, please look at the included PoC of the translator component.

The intention of the translator component is to be small and lightweight so that we can scale by adding additional translators to process work as it builds up in the queue.

Performance Considerations

Key Testing Considerations

- Translator consumers – Since this component is critical to the system it's important that it be unit tested thoroughly. In particular key areas to be tested are:
    - Parsing of JSON mappings – The brunt of the actual parsing will be handled by our JSON Object mapping library, but it's important to test error conditions (malformed JSON, invalid data types, missing attributes, impossible column mappings, etc.)
    - CSV parsing – Here again, the libraries are doing the heavy lifting for actual processing of the CSV files, but we need to make sure error conditions like empty files, malformed files, etc. are handled gracefully.
    - Data Transformation – We need to make sure that our mappings are being applied correctly and that test output matches what was expected. Validations also need to be checked to ensure that they are being applied correctly and that error output is being correctly captured and processed.
- UI – Selenium testing needs to be done, particularly around the mapping UI. We need to ensure correct JSON is being generated and that the UI does not allow the user to enter invalid mappings. Additional tests need to be written around file uploading/downloading, and display/editing of CSV data.
- Message routing – Integration tests are needed to ensure that messages are being routed correctly and that responses in the RPC model are being sent back to the correct response queue for processing by the UI. We also need tests to ensure that any sort of invalid messages are routed to an error queue so that they can be investigated.

Performance Monitoring

Monitoring application performance is important to ensure a good user experience and to identify potential problems as early as possible. There are a few key areas that should be monitored to give us a good view into the performance health of the application:

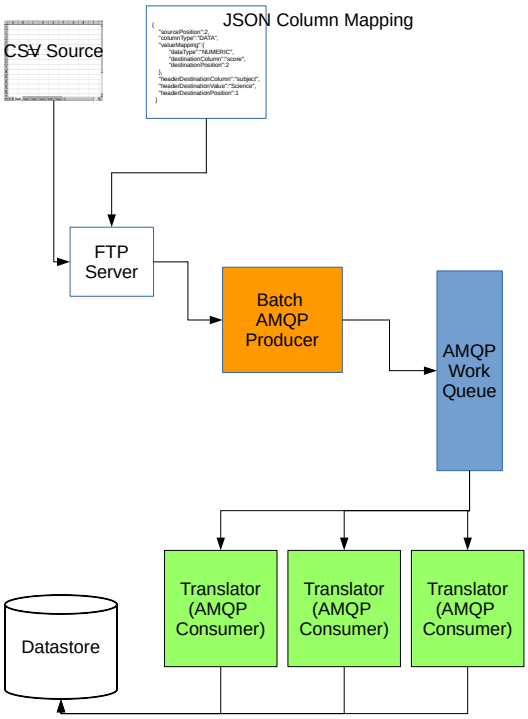
- Message processing time – Measuring the average time taken for a message to be placed on the queue, and for a response to be returned gives us a good baseline performance indicator. If round-trip processing times begin to climb consistently above the average then it may indicate a performance issue.
- Queue depth- Monitoring the number of messages in the queue is a good way to keep an eye on the health of the translators and datastore. If the depth of the queue begins increasing significantly it may indicate a performance problem or even an outage.

Potential System Bottlenecks

One potential issue with using the RPC setup for messaging, is that the application will block while waiting for the data transformation to complete and for the response to be sent back to the web application. This means that there is a potential bottleneck while waiting for large datasets to process. We can keep an eye out for this issue by monitoring the average message processing time as described above. If performance issues are detected, limiting the size of datasets uploaded through the UI should help prevent long response times.

As an alternative to processing large files through the web interface, providing a batch processing option for handling especially large datasets should improve response times. The UI can still be used for creation and export of the JSON column mappings, which will then be uploaded with the large CSV files for batch processing. Batch processing can be further optimized by modifying the AMQP messaging strategy to use a simpler work queue strategy that does not send a response back to the client. This allows us to split the dataset into smaller chunks and have multiple translators working on

the data in parallel.



Scaling

Using message queues to handle routing requests means that no additional configuration or load balancing is needed to add additional translators. This allows us to scale the solution easily by adding additional translators to improve throughput when faced with heavy traffic.