

# Lesson 29: CSV Batch Processing and Exception Handling

Created by Charl Viljoen, last modified on Jul 17, 2020

*As a Shop Owner I want to do bulk updates to my stock levels by uploading CSV files, however I want every line to be added regardless of whether other lines have failed.*

- [Lesson Outcomes](#)
- [New & Modified App Files](#)
- [Standard CSV Processing vs Batch CSV Processing](#)
- [Updating the old CSV upload](#)
- [Exception Handling](#)
- [Further Reading](#)
- [Lesson Source Code](#)

## Lesson Outcomes

By the end of this lesson you should:

- Know how use CSV batch processing to process each line an a CSV file individually
- Know how to use exception handling in the DSL to prevent a single invalid line in a CSV file from preventing the entire file from being processed.


## New & Modified App Files

```
./model/objects/CsvException.mez
./web-app/lang/en.lang
./web-app/presenters/StockLevelsUpdate.mez
./web-app/views/StockLevelsUpdate.vxml
```

## Standard CSV Processing vs Batch CSV Processing

Standard CSV Processing parses the entire blob object that represents the .csv file and results in a collection of object instances. If one invalid line is present in the CSV file, the entire process would halt and throw an error. This provides an added layer of protection when uploading a large amount of data that relates to each other. Everything is therefore dependent that everything else in the file processes correctly.

Batch CSV Processing, however, parses each line in the .csv file individually allowing developers to add their own exception handling for the cases when a single line fails. This is more applicable to files where each line represents its own individual identity and can be processed regardless of the status of the other lines in the file.


 See [here](#) for more information on CSV Processing in Helium.

## Updating the old CSV upload

To demonstrate the difference between how CSV files are processed in Helium, we're just going to update the CSV processing already put in place during [Lesson 9](#). We will also use some exception handling to illustrate how developers can have more control over their applications by allowing the program to still flow in a logical fashion even when an exception might be encountered.

While the Standard CSV Processing uses only one step, one built-in function (the `fromCsv()` function) that process the entire CSV file, Batch CSV Processing uses multiple steps.

- First the `Mez:createBatch` built-in function is used to create a batch from a blob representing a valid CSV file.
- The built-in `MezBatch` and `MezBatchItem` objects are used to store the created batch.
- The created batch can then be processed item by item using the `fromCsvLine()` built-in function.

 **VERY IMPORTANT**  
These objects are persistent. Creating them will persist to the database.

If you expect/do regular csv uploads on your application please take note that these **MezBatch** and **MezBatchItem** records will be added to you schema.

It is advisable that you clear these tables often to ensure your schema doesn't grow needlessly large.

We're going to update the `saveStockUpdate()` function in the `StockLevelsUpdate.mez` file as follows:

presenter snippet (StockLevelsUpdate.mez)

```

1  void saveStockUpdate() {
2      //Create a batch from the uploaded csv file
3      MezBatch stockUpdateBatch = Mez:createBatch(fileUpload.
4
5      // Get the batch items related to the batch being proce
6      MezBatchItem[] stockUpdateBatchItems = MezBatchItem:rel
7
8      // Results from CSV processing will be stored in this c
9      StockUpdate[] stockUpdates;
10
11     // Iterate over the batch items and process one by one
12     foreach(MezBatchItem stockUpdateBatchItem: stockUpdateB
13         //Create a stockUpdate instance for each MezBatchIt
14         StockUpdate stockUpdate = StockUpdate:fromCsvLine(s
15         stockUpdate.stocktakeDate = selectedDateOfStocktake
16         stockUpdate.shop = selectedShop;
17         stockUpdate.stock = getStockFromName(stockUpdate.st
18         stockUpdate.save();
19         stockUpdates.append(stockUpdate);
20
21         // Once a batch item has been successfully processe
22         stockUpdateBatchItem.processed = true;
23     }
24     Mez:alert("alert.uploaded_data_saved");
25     fileUpload = FileUpload:new();
26 }

```

As described before, we first create a `MezBatch` instance from the uploaded file which creates a `MezBatchItem` instance for every line in that file and has a relationship to the created `MezBatch` (go read more [here](#) about these objects and Batch CSV Processing). Secondly, we populate a list of `MezBatchItem` instances by finding those with a relationship to the created `MezBatch`. Thirdly, we loop over this list of `MezBatchItem` instances and process each one with the `fromCsvLine()` function, contrary to before where we used the `fromCsv()` function to process the entire file.

This works just fine, but doesn't leverage the usefulness of processing each line individually and finding or handling issues or exceptions.

## Exception Handling

Often developers can predict scenarios which might cause issues or errors in their programs and they will attempt to avoid these scenarios as much as possible. However, sometimes they are unavoidable and even very likely to happen but you'd rather not have the entire application crash as a result. Exception handling enables programmers to cater for these scenarios and override the normal program flow during runtime when these exceptions occur. You can find more about Exception Handling in Helium [here](#) but in this lesson we're going to stick to a simpler example.

When trying to process each `MezBatchItem` instance in the function above, we want to catch any issues that might occur as Helium attempts to create a `StockUpdate` instance using the `fromCsvLine()` function. Helium provides exactly this functionality in a `Try Catch Block` which allows the other instances to continue processing without halting the entire application in the middle of processing the CSV file.

presenter snippet (StockLevelsUpdate.mez)

```

1  void saveStockUpdate() {
2      MezBatch stockUpdateBatch = Mez:createBatch(fileUpload.
3
4      // Get the batch items related to the batch being proce
5      MezBatchItem[] stockUpdateBatchItems = MezBatchItem:rel
6
7      // Results from CSV processing will be stored in this c
8      StockUpdate[] stockUpdates;
9
10     // Iterate over the batch items and process one by one
11     foreach(MezBatchItem stockUpdateBatchItem: stockUpdateB
12         try{
13             StockUpdate stockUpdate = StockUpdate:fromCsvLi
14             stockUpdate.stocktakeDate = selectedDateOfStock
15             stockUpdate.shop = selectedShop;
16             stockUpdate.stock = getStockFromName(stockUpdat
17             stockUpdate.save();
18             stockUpdates.append(stockUpdate);
19
20             // Once a batch item has been successfully proc
21             stockUpdateBatchItem.processed = true;
22         }
23         catch(exception) {
24             // Handle the exception by logging it and creat
25             handleException(stockUpdateBatch, stockUpdateBa
26         }
27     }
28     Mez:alert("alert.uploaded_data_saved");
29     fileUpload = FileUpload:new();
30 }

```

This Try Catch Block will attempt to complete everything in the `try{}` code block from top to bottom, and if it encounters any Helium exceptions, it will jump immediately to the `catch{}` block to continue with processing until the entire function is resolved. Please note that the `handleException()` function in the `catch{}` block above has not yet been declared. Also that the `exception` caught in the `catch{}` block has several values that can be used and here we are using the `message` value.

For our application we would like to display any exceptions that might occur to the Shop Owner so that they can rectify it and upload these corrections in a new csv file. We're going to give them a message explaining just that and show a table indicating exactly what was the exception. First we'll add a non-persistent object to the model that can logically house these exceptions temporarily as we don't want to keep these messages for use outside of their session.

#### CsvException object

```

1  object CsvException {
2      string header;
3      string value;
4      string exception;
5  }

```

Then we can declare the `handleException()` function implemented in the `catch{}` block above, also adding a list (`unprocessedItems`) to the presenter that will house these exceptions and a

boolean value (`showExceptions`) to toggle the table in the view which shows these exceptions to the user.

#### presenter snippet (StockLevelsUpdate.mez)

```

1  unit StockLevelsUpdate;
2
3  CsvException[] unprocessedItems;
4  bool showExceptions;
5
6  // Init function to initialize unit variables
7  void init() {
8      showExceptions = false;
9      ...
10 }
11
12 void saveStockUpdate() {
13     unprocessedItems.clear();
14     ...
15     if (unprocessedItems.length() > 0) {
16         Mez:alert("alert.uploaded_data_saved_with_exc");
17         showExceptions = true;
18     } else {
19         Mez:alert("alert.uploaded_data_saved");
20         showExceptions = false;
21     }
22     fileUpload = FileUpload:new();
23 }
24 ...
25
26 void handleException(MezBatch updateBatch, MezBatchItem upd
27     //Mark that the MezBatchItem has not been processed suc
28     updateItem.processed = false;
29     //Log the exception
30     Mez:log(exception);
31     //Add to exception list to display to User
32     CsvException item = CsvException:new();
33     item.header = updateBatch.header;
34     item.value = updateItem.value;
35     item.exception = exception;
36     unprocessedItems.append(item);
37 }
```

The view just gets an added table that displays the `unprocessedItems` when the `showExceptions` is true.

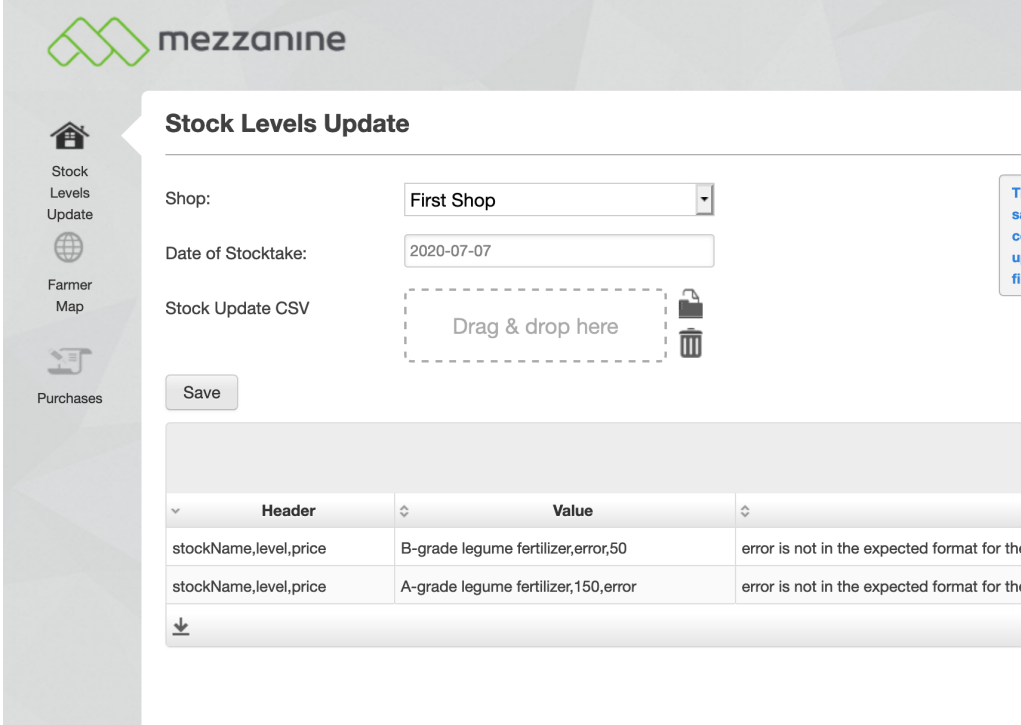
#### view snippet (StockLevelsUpdate.vxml)

```

1
2 <table>
3     <visible variable="showExceptions"/>
4     <collectionSource variable="unprocessedItems"/>
5     <column heading="column_heading.header">
6         <attributeName>header</attributeName>
7     </column>
```

```
8 <column heading="column_heading.value">
9   <attributeName>value</attributeName>
10 </column>
11 <column heading="column_heading.exception">
12   <attributeName>exception</attributeName>
13 </column>
</table>
```

This would be an example of what your application should look like then.



As can be seen in the values, the user attempted to add values for level and price which could not be formatted to the expected column type. Instead of the whole csv processing throwing an error, only two lines failed and have to be corrected before uploading again while the other 8 were successfully persisted. Remember to update the `en.lang` file with the new translations and see the attached source code for any omitted parts if necessary.

### Further Reading

Read more about CSV Processing here: [CSV Processing](#), specifically [Batch CSV Processing](#)

Read more about Exception Handling here: [Exception Handling](#)

### Lesson Source Code

[Lesson 29.zip](#)

No labels