

Data/Record/Column validation

From release 2020.09 on Servoy has added/changed how validation is done for database data.

databaseManager has now a new method: JSRecordMarkers databaseManager.validate(record [, optionalState]);

Which is automatically called when saveData() is done for every record that was changed and will be saved to the database. But you can also call it manually your self to report possible validation problems.

It will return a JSRecordMarkers when the record has validation problems, if everything was ok then a null is returned.

The optionalState object can be anything that you want to pass on to all of the validators that are being called.

The validate(record) call will first call the new "onValidate" table event method which has the signature: (record, recordMarkers, stateObject).

- record: the record to validate.
- recordMarkers: the JSRecordMarkers where problems can be reported in to
- stateObject: is the optionalState object you give to the databaseManager.validate(record, {something:1}) call for extra information that you want to give into the onValidate entity event.

If there is a problem that should be reported the JSRecordMarkers has a few report methods that you can use to report problems, the largest one has the signature: (message, dataprovider, level, customObject, messageKeyParams), only message is mandatory, the rest are optional.

- message: the message your want to report back to the user (can be an i18n key then it should start with "i18n:")
- dataprovider: the column where this message belongs to
- level: a LOGGINGLEVEL level, so you can report that this is really an ERROR or just INFO/WARNING.
- customObject, the same as the optionalState object this can be anything that can be used later on the have more information about this problem.
- messageKeyParams: if the message was an i18n key that has dynamic values then this is the array that has those dynamic values that are injected into the final message.

After the "onValidate" table event, it will call one of the "onRecordUpdate" or "onRecordInsert" table event depending on if the record is an existing or new record. The signature of those table events are also changed: (record, recordMarkers, stateObject), so those are now the same as "onValidate". Instead of returning false or throwing an exception it is now better to just report problems on the recordMarkers and always just return true. This way you control the message.

The 3rd step is to check if there are columns that are null or an empty string but the column is marked as not null in the database and it doesn't have a default database value, which would be filled in when we insert. If the column has this problem it will be reported with the i18n key: servoy.record.error.null.not.allowed ("Dataprovider '{0}' can't be null or empty"), also an exception object is set on the Record.exception to be more in line what did happen before.

The 4th step is to check the size of the column of only the changed columns, if a column has a value that won't fit into the database it will be reported by the i18n key: servoy.record.error.columnSizeTooSmall ("Column '{0}' is too small, max length can be {1}, value '{2}'")

The last step is calling all the [Column Validation](#) for all the changed columns of the record. By default Servoy will not call the column validators anymore when the value is set into the record, this could be a **breaking change** for some scenarios that depend on this, it will only validate the columns when saving or when the validate(record) method is called. This change of behavior is controlled by the servoy property "servoy.execute.column.validators.only.on.validate_and_save" which is default true. In the table editor column validation section there is now a check box that controls this property "Only execute validators on validate/save". But by default now the Record could have invalid data in memory, previously Servoy wouldn't allow this.

All the build in validators are changed to use the new way, this means for the GlobalMethodValidator that also there the signature is changed from just getting the value object to: (value, dataproviderid, recordMarkers, stateObject)

- value: the value to validate.
- dataproviderid: the is the dataprovider that is being validated, can be used in a report call so it is known which dataprovider/column had this problem.
- recordMarkers: the JSRecordMarkers where problems can be reported in to
- stateObject: is the optionalState object you give to the databaseManager.validate(record, {something:1})

Also for this the same applies as from the "onRecordUpdate" instead of return false or throwing an Exception, the global method validator should report problems to the recordMarkers and always just return true.

All of the steps are called, even if one already fails, so the JSRecordMarkers can have multiply reported problems at once. When all the steps/checks are done and it has at least 1 problem reported then it will set itself on the JSRecord.recordMarkers property (which is cleared before any validate) and the recordMarkers will be returned.

If the caller of databaseManager.validate(record) sees that an object is returned it can then call recordMarkers.getMarkers() to get an array of JSRecordMarker. A JSRecordMarker has has a number of fields:

- message: the message that was reported by the call JSRecordMarkers.report(message), can just be an i18n key
- i18NMessage: the resolved message if the above message was an i18n key.
- column: the column for which this problems was generated
- level: the LOGGINGLEVEL of this problem.
- customObject: the custom object that was passed into the report() method, for standard or build in checks like the null check of the RangeColumnValidator this custom object is the object that you give into the validate(record, x) call.
- record: the record for which this problems was generated

```

var recordMarkers = databaseManager.validate(record, {originationCall: "customerForm"});
if (recordMarkers)
{
    var markers = recordMarkers.getMarkers();
    markers.forEach(** @param {JSRecordMarker} marker*/ function(marker) {
        application.output(marker.message)
        application.output(marker.i18NMessage);
        application.output(marker.column);
        application.output(marker.customObject);
    });
}

```

Besides the `getMarkers()` the `JSRecordMarkers` has also some extra state, 2 boolean properties: `onBeforeInsertFailed` and `onBeforeUpdateFailed` those are filled in if the `onRecordUpdate` or `Insert` failed in a legacy way (returning false), here is also a `getGenericExceptions()` method that will return exception that could be happen in the validation code that are not directly record or column validation problems but more generic once like exceptions that are triggered when the `onRecordUpdate` or `Insert` did fail.

Saving

if you call just `databaseManager.saveData([record])` then that will also call `validate` for all the records that are being saved. The save of a record will be blocked when at least one validation problem is reported with the `LOGGINGLEVEL` of `ERROR` or `FATAL`. The save will just go on when only `WARN` or `INFO` is used. This can also be asked after validation: `JSRecordMarkers.hasErrors` gives back a boolean if this really has errors that blocked the save.

If the save is blocked by validation errors then the record will be put into the `failedRecords` list (and removed from the `editRecord` list), all records will be processed, so the first will not directly return false for the `saveData()`.

When the record is validated and the database is called to insert or update a database exception can happen, this is still set on the `Record.exception`, but also a `JSRecordMarkers` is created and the exception is added to the `genericExceptions`.

Autosave and failing to save (validation or db exceptions)

When autosave is on (see also [Side Effects](#) below) Servoy tries to save at certain points the records that are edited. Those points can be things like record selection changes (then the foundsets edited records are tried to be saved) or a form is made invisible, or the user clicks on an empty space on the form.

When validation fails or the database fails then a new method on the solution is called: `onAutoSaveFailed(JSRecordMarkers[])` with all the failed records record markers. This is like the `onError` callback (Which is still also called for actual exceptions in the save) but very specific for the saving of record in autosave mode. You could for example add logging there so then you see why certain actions are blocked (like going to another form).

This Solution event is only called for autosave, so any javascript triggered validation or `saveData()` will not result in that method to be called, then the `validate` or `saveData()` will already return the state that it failed and you can process at that time the `failedRecords`.

Side effects

Because the column validator doesn't run by default anymore when the value is pushed into the record, and an illegal value is blocked previously, now the illegal value will be placed in the record. And the record won't be able to save itself. This can result in a form not allowing itself to go invisible. Because a form tries first to save all the records of its foundset that are in edit when hiding the form. If autosave is on the records are tried to save and fail now so the form stays visible.