Managing concurrent updates on Phonograph

Kai Altstaedt

Tutorial / Implementation guide

http://www.lambdasign.de

Abstract

The Objects/Ontology layer is the recommended product for writeback scenarios in the Palantir Foundry. But for data assets, that will not get their own Object, a writeback using Phonograph as a datastore is still a suitable way. Using native Phonograph, the problem of managing concurrent access or more precise concurrent updates on data may arise.

This tutorial shows a way how to use the Phonograph API in a Slate application to manage the concurrent access while providing a good user-experience. The core of the solution is to use the Version of the data of a row, which is maintained by Phonograph in the “editsVersion” of a row. Using the “editsVersion” concurrent updates can be identified and managed.

Advanced knowledge of Slate and basic knowledge of interacting from Slate with Phonograph is assumed.

Level: Advanced Slate Programming

Keywords: Foundry, Slate, Phonograph

Managing concurrent updates on Phonograph

The Foundry has a natural tendency to move from a data-analytics platform with the main focus on Information retrieval and analytics, to an element of an enterprise architecture, where the "leading data" is stored.

This starts typically with a pattern to comment/enrich results of the analyses or an user-status and moves to applications inside of the Foundry with a full CRUD loop to "Create", "Read", "Update" and "Deleted" data. The more operational the data becomes, the higher the need to care for concurrent updates is typically.

The Object layer as the recommended architecture pattern for a writeback of data provides means to deal with the update, but in various situations and depending on the Foundry setup, the Foundry technology "Phonograph" is still in use to implement writebacks. This tutorial shows ways to manage concurrent updates. It was triggered by the blogpost "Rethinking CRUD for REST APO" from David Xiao, Software Engineering Lead at Palantir (David Xiao, 2021). A great thanks to Logan Rhyne who guided me through some challenges during implementation.

About the author

Kai Altstaedt works as an IT specialist at a large Aircraft manufacturer, free author, programmer, maker and manufacturer. He published some articles on various IT topics for German IT magazines and is the author of the book "Palantir Foundry by Use cases" (Altstaedt, 2021).
From his IT heritage he has a strong focus on programming, databases and machine learning. Programming is his great passion.
More information can be found at www.lambdasign.de

Framing

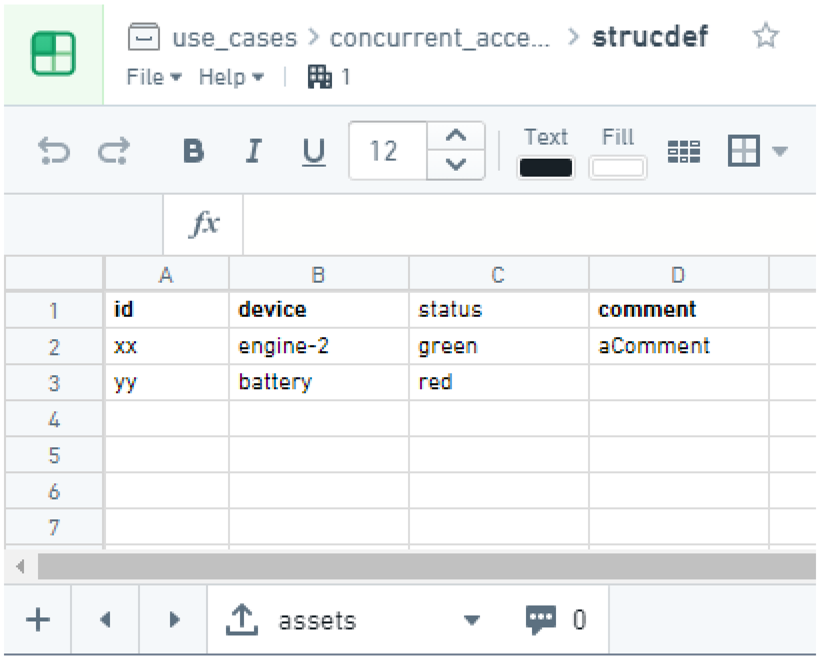
Use Case

As an example, a use case from a manufacturing area is taken. There are two assets: One engine and one battery. In the status database, both of them have a status and comment field. The teams monitoring the assets are spread over various locations and work in parallel on the status of a device. So it is quite likely, that concurrent updates may happen.

Basic architecture

Phonograph is used as the persistence layer. Slate for the Application Layer.

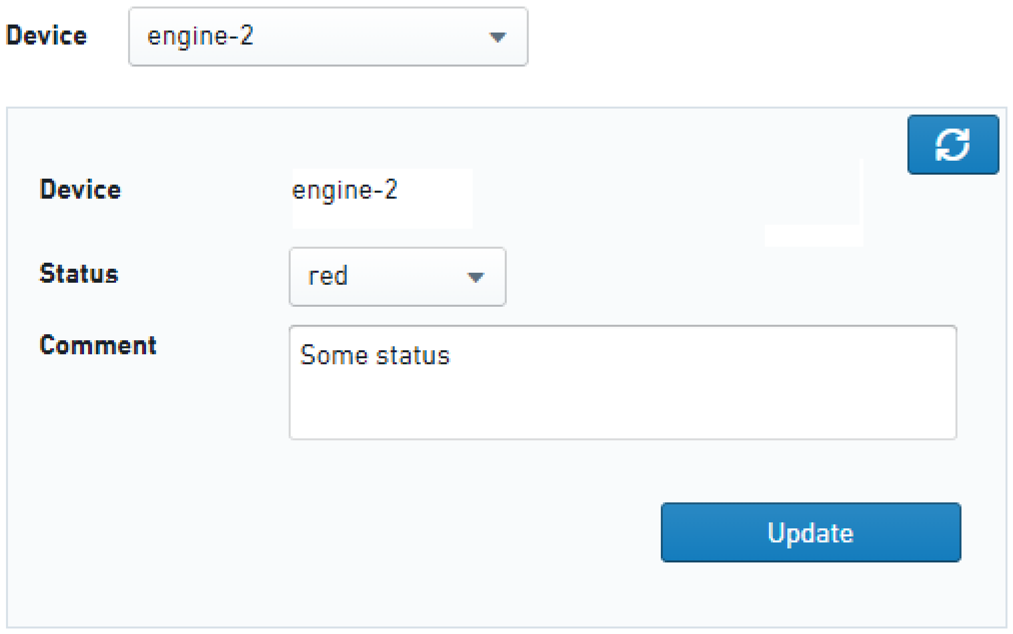
The schema of the data and the available assets are set up in a Fusion table that is synchronized to Phonograph.



	A	B	C	D
1	id	device	status	comment
2	xx	engine-2	green	aComment
3	yy	battery	red	
4				
5				
6				
7				

Structure and content definition

The UI is implemented in Slate. Filling of the Dropdown and the Input Fields is plain vanilla Phonograph and Slate. When the “Update” Button is hit, the data is written back to the phonograph index.


Device: engine-2

Device: engine-2

Status: red

Comment: Some status

For simplicity only an update is implemented. The Update is done with through the postEvent of Phonograph.

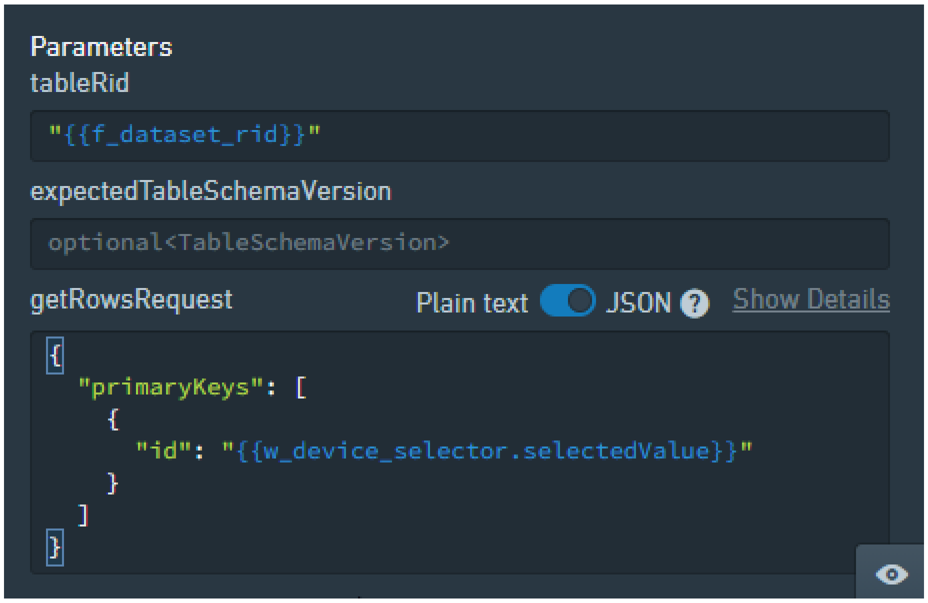
Version 1 – Optimistic concurrency with simple failure indication

The first implementation approach makes a basic use of the Phonograph API to detect concurrent updates. Reading carefully the API specification of the "getRows" and "PostEvent", one discovers the attribute *editsVersion*.

This attribute contains the version of the rowData as a long-Value. Playing around with it shows that it does, what the API tells: It is a number, increasing by 1 with each update.
The post-event allows to pass an expected *editsVersion*. The first implementation makes basic use to the editsVersion to identify concurrent updates.

Reading and remembering of the editsVersion

When the user selects an asset in the dropdown, a query to retrieve the details is launched.



tableRid

```
"{{f_dataset_rid}}"
```

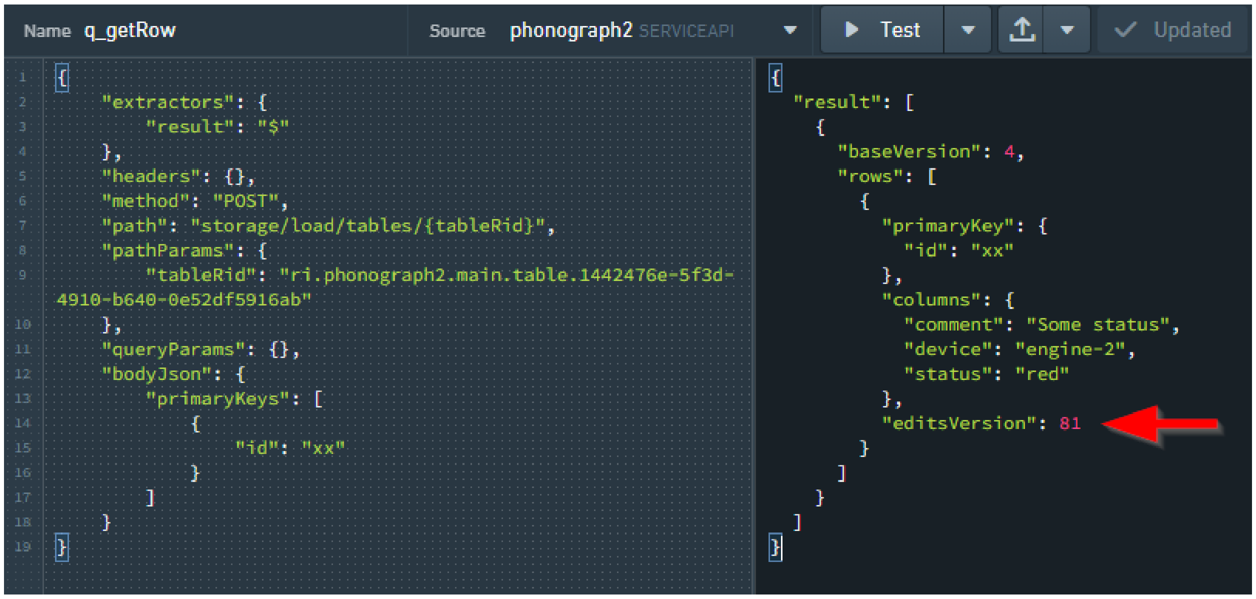
expectedTableSchemaVersion

```
optional<TableSchemaVersion>
```

getRowsRequest

```
{
  "primaryKeys": [
    {
      "id": "{{w_device_selector.selectedValue}}"
    }
  ]
}
```

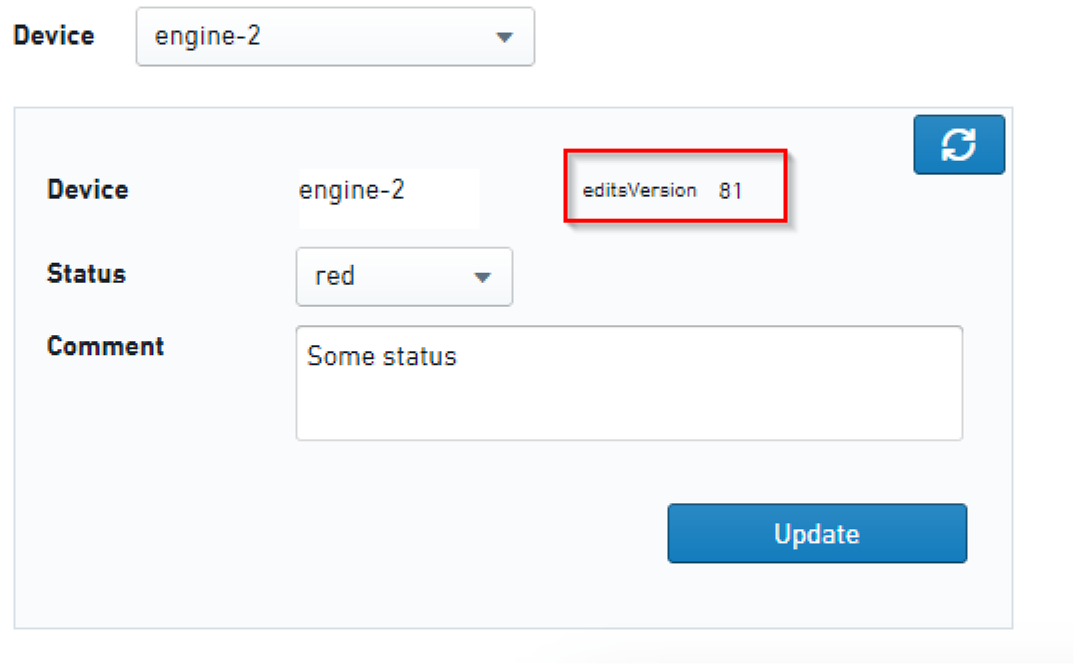
The result does not only contain the column data but as well per row the editsVersion.



```
{
    "extractors": {
        "result": "$"
    },
    "headers": {},
    "method": "POST",
    "path": "storage/load/tables/{tableRid}",
    "pathParams": {
        "tableRid": "ri.phonograph2.main.table.1442476e-5f3d-
4910-b640-0e52df5916ab"
    },
    "queryParams": {},
    "bodyJson": {
        "primaryKeys": [
            {
                "id": "xx"
            }
        ]
    }
}
```

```
{
  "result": [
    {
      "baseVersion": 4,
      "rows": [
        {
          "primaryKey": {
            "id": "xx"
          },
          "columns": {
            "comment": "Some status",
            "device": "engine-2",
            "status": "red"
          },
          "editsVersion": 81
        }
      ]
    }
  ]
}
```

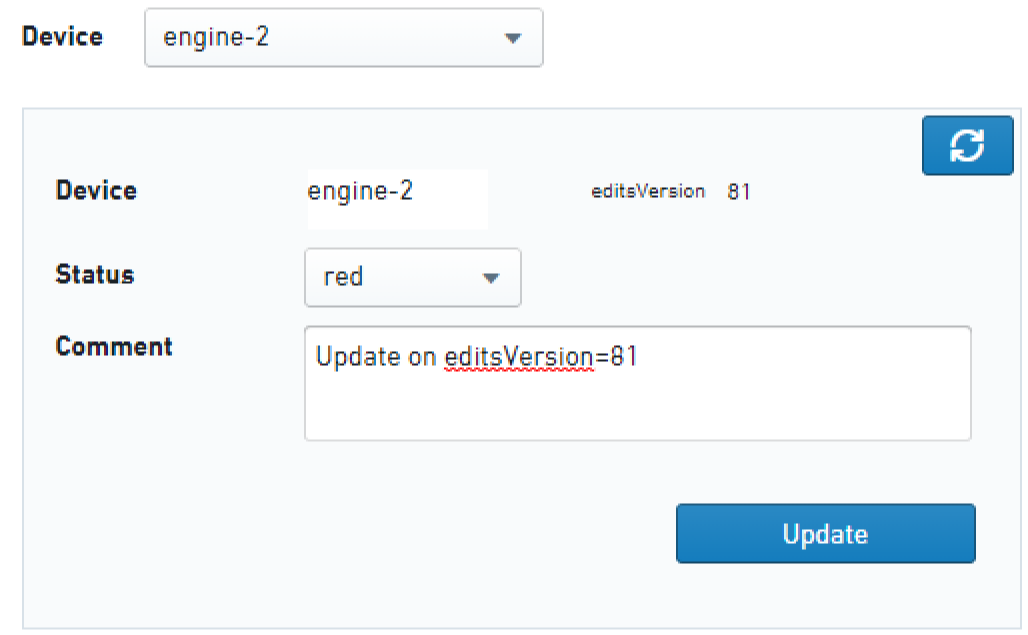
Both the data and the editsVersion are pushed to the UI.



Device	engine-2
Device	engine-2
editsVersion	81
Status	red
Comment	Some status

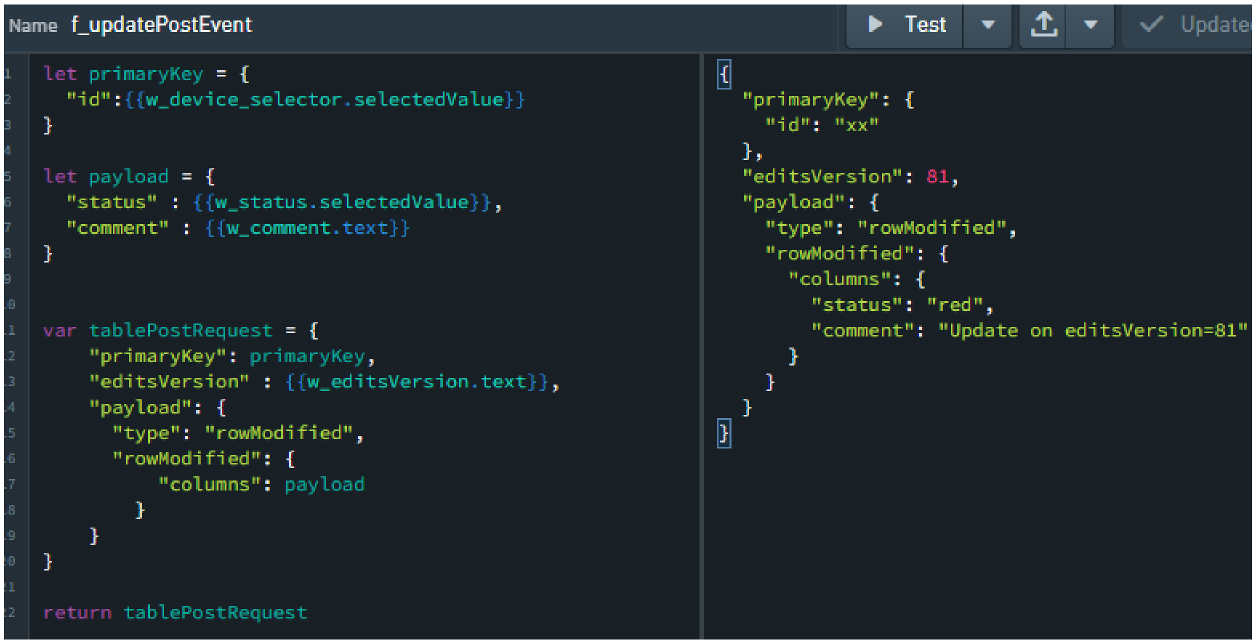
Performing an Update

Some new information is then entered to the input Field for the comment.



Device	engine-2	
Device	engine-2	editsVersion 81
Status	red	
Comment	Update on editsVersion=81	

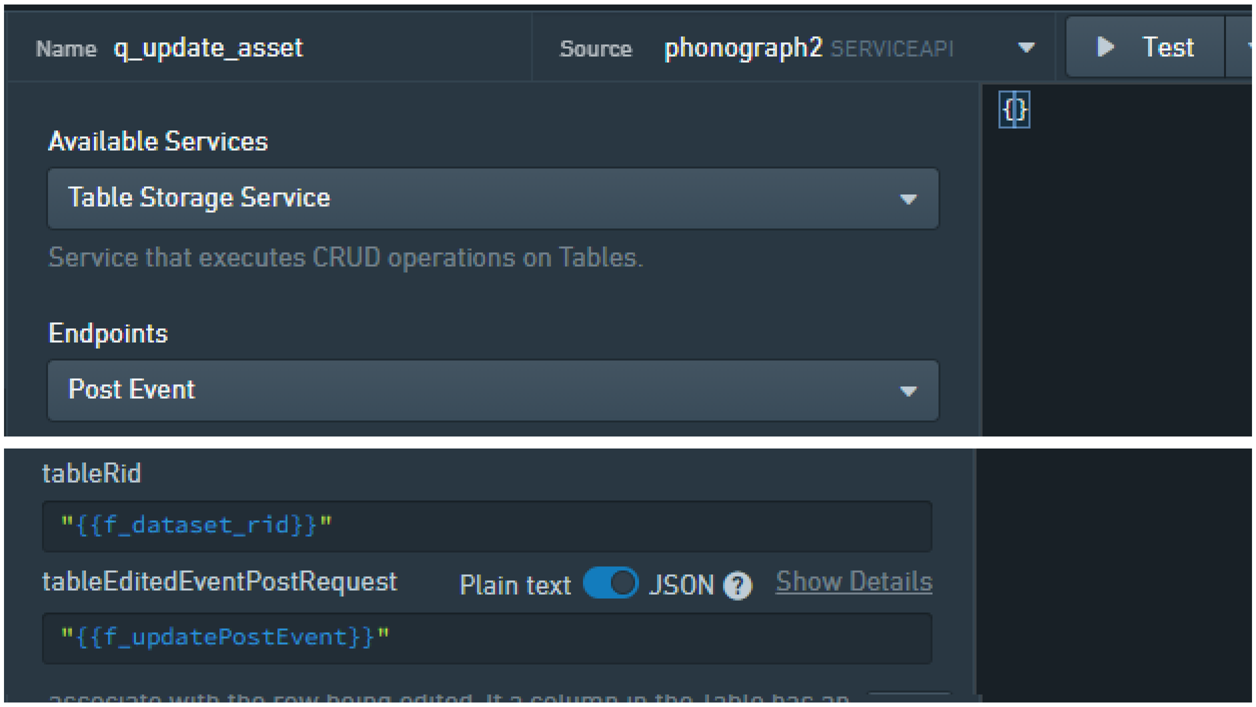
For an update, the optional request attribute *editsVersion* is pushed to the update request.



```
let primaryKey = {
  "id":{{w_device_selector.selectedValue}}
}

let payload = {
  "status" : {{w_status.selectedValue}},
  "comment" : {{w_comment.text}}
}


var tablePostRequest = {
    "primaryKey": primaryKey,
    "editsVersion" : {{w_editsVersion.text}},
    "payload": {
      "type": "rowModified",
      "rowModified": {
          "columns": payload
        }
    }
}

return tablePostRequest
```

```
{
  "primaryKey": {
    "id": "xx"
  },
  "editsVersion": 81,
  "payload": {
    "type": "rowModified",
    "rowModified": {
      "columns": {
        "status": "red",
        "comment": "Update on editsVersion=81"
      }
    }
  }
}
```

The query is executed



Name q_update_asset

Source phonograph2 SERVICEAPI

Test

Available Services

Table Storage Service

Service that executes CRUD operations on Tables.

Endpoints

Post Event

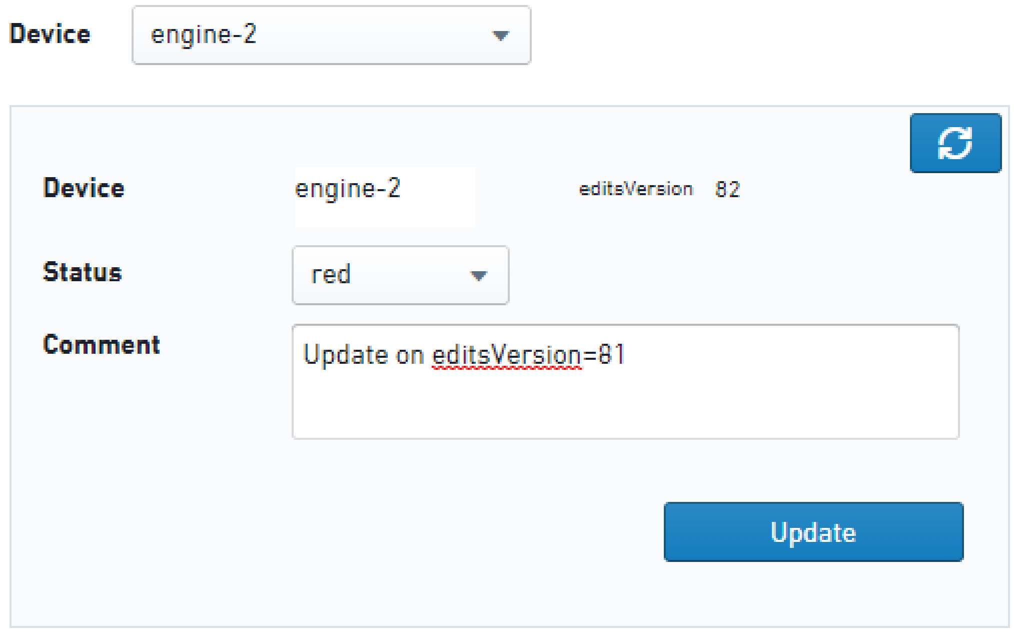
tableRid

"{{f_dataset_rid}}"

tableEditedEventPostRequest Plain text JSON Show Details

"{{f_updatePostEvent}}"

With the usual event->toast->reload pattern the UI is refreshed:



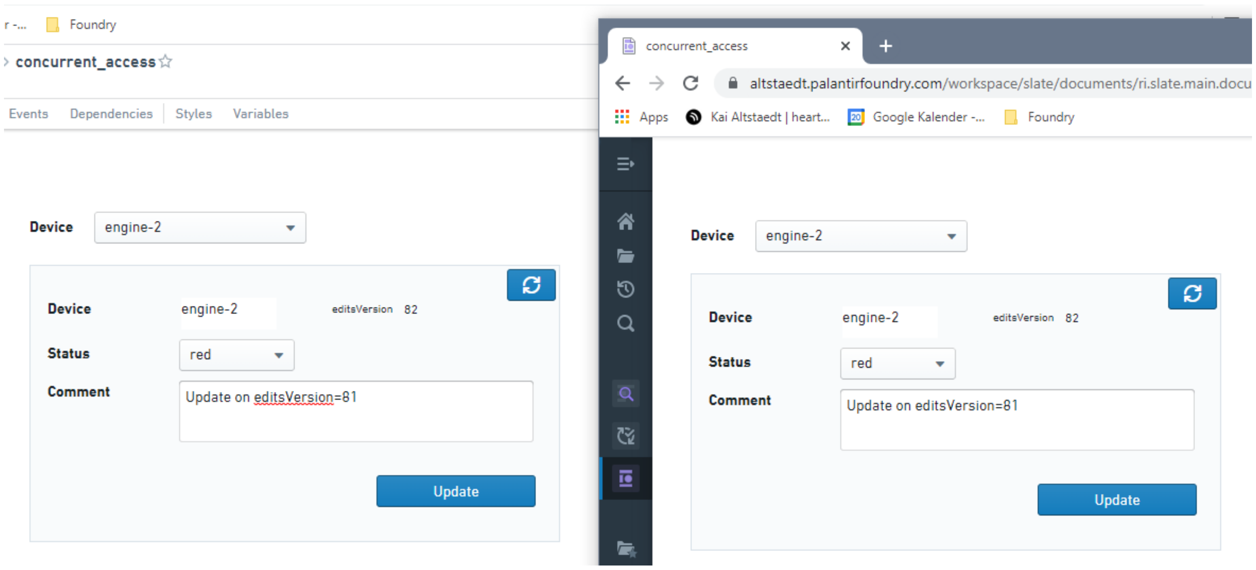
Device engine-2

Device	engine-2	editsVersion	82
Status	red		
Comment	Update on editsVersion=81		

The update on the row with the editsVersion=81 resulted in a row with the version 82. The update and behavior of the editsVersion works as expected.

Performing a concurrent update

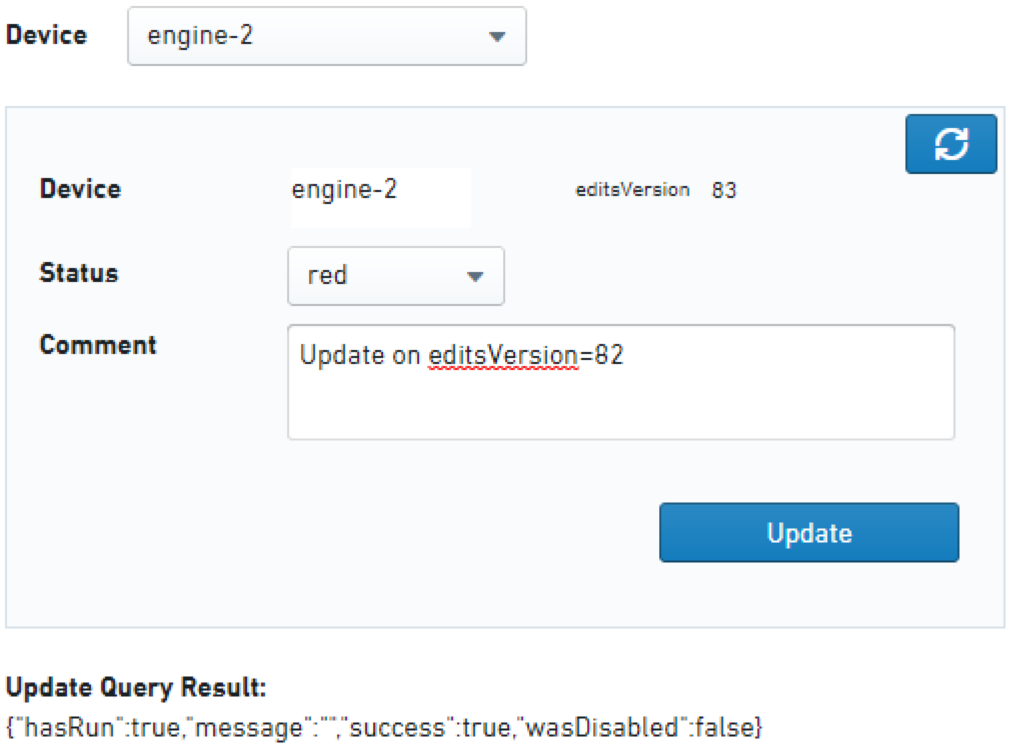
A second browser window is now opened with the same application.



Device: engine-2
Device: engine-2 editsVersion 82
Status: red
Comment: Update on editsVersion=81

Device: engine-2
Device: engine-2 editsVersion 82
Status: red
Comment: Update on editsVersion=81

In the second window an update is performed (without touching the first one).

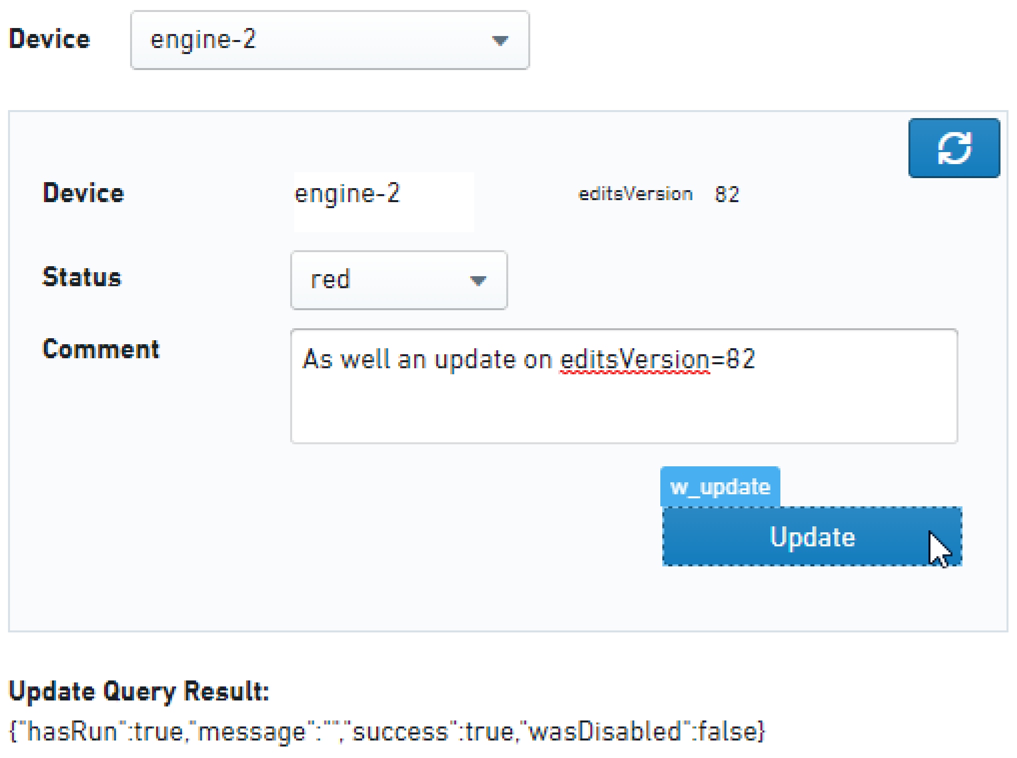


Device: engine-2
Device: engine-2 editsVersion 83
Status: red
Comment: Update on editsVersion=82

Update Query Result:

```
{"hasRun":true,"message":"","success":true,"wasDisabled":false}
```

We get Version 83. The query result shows no problems. Hopping now back to the other browser window still having editsVersion 82 active, we edit again some data…..



Device engine-2

Device engine-2 editsVersion 82

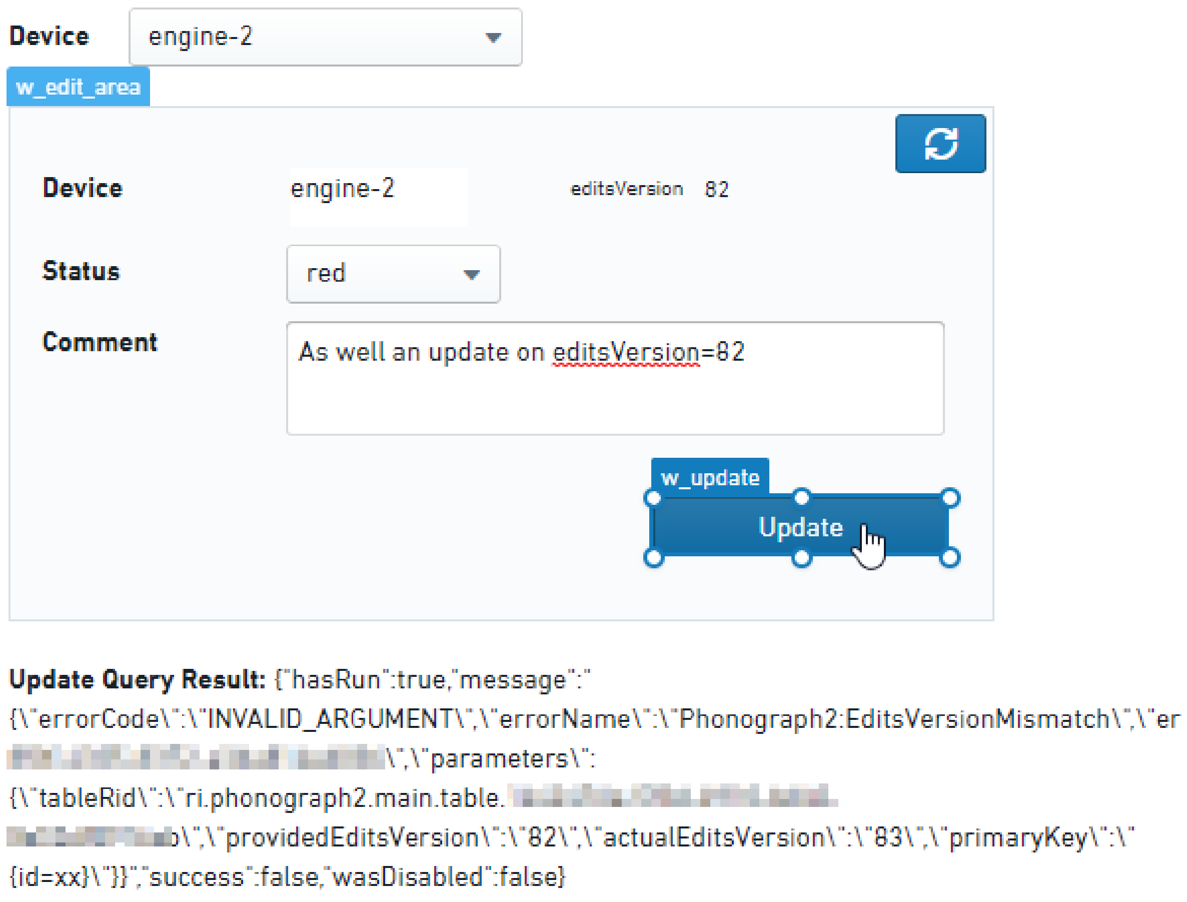
Status red

Comment As well an update on editsVersion=82

Update Query Result:

```
{"hasRun":true,"message":"","success":true,"wasDisabled":false}
```

And hit the Update for a concurrent update…

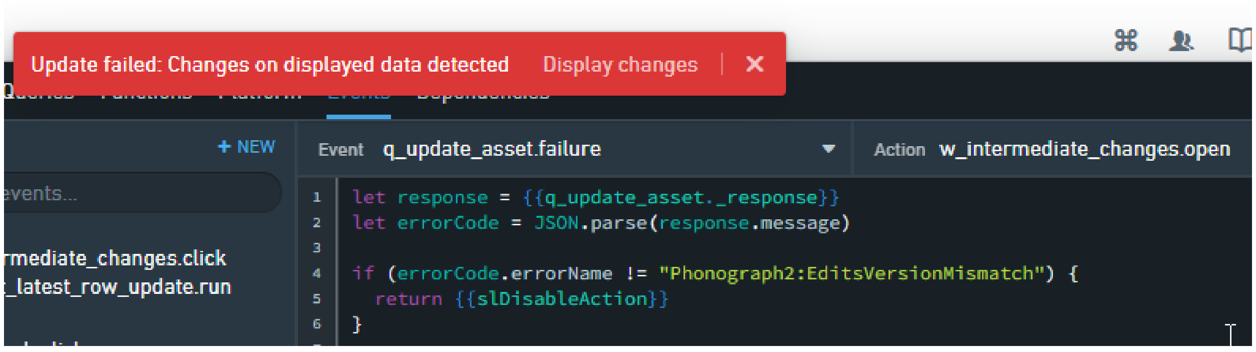


Device	engine-2
editsVersion	82
Status	red
Comment	As well an update on editsVersion=82

Update Query Result: {"hasRun":true,"message":"
{\"errorCode\":\"INVALID_ARGUMENT\",\"errorName\":\"Phonograph2:EditsVersionMismatch\",\"er
[redacted]\",\"parameters\":
{\"tableRid\":\"ri.phonograph2.main.table.[redacted]
[redacted]b\",\"providedEditsVersion\":\"82\",\"actualEditsVersion\":\"83\",\"primaryKey\":\"
{id=xx}\"}}","success":false,"wasDisabled":false}

Very well! In the meantime, the version of the row switched and since the expected editsVersion did not match the current editsVersion in the database the update failed.

Examining the errorName, we can catch this particular situation and launch a toast.

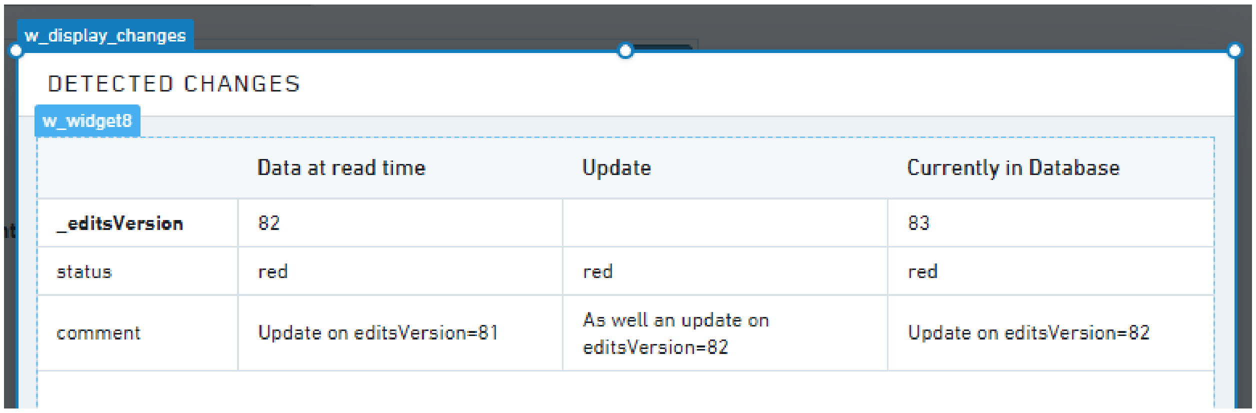


Event q_update_asset.failure
Action w_intermediate_changes.open

```
let response = {{q_update_asset._response}}
let errorCode = JSON.parse(response.message)

if (errorCode.errorName != "Phonograph2:EditsVersionMismatch") {
  return {{slDisableAction}}
}
```

Catching this action, a second query can be launched, that retrieves the current version from the backend. The action of the toast launches a dialog that displays the three versions of data. The three versions are:

1. Data at the time of reading
2. Updates from the UI
3. Version in the database



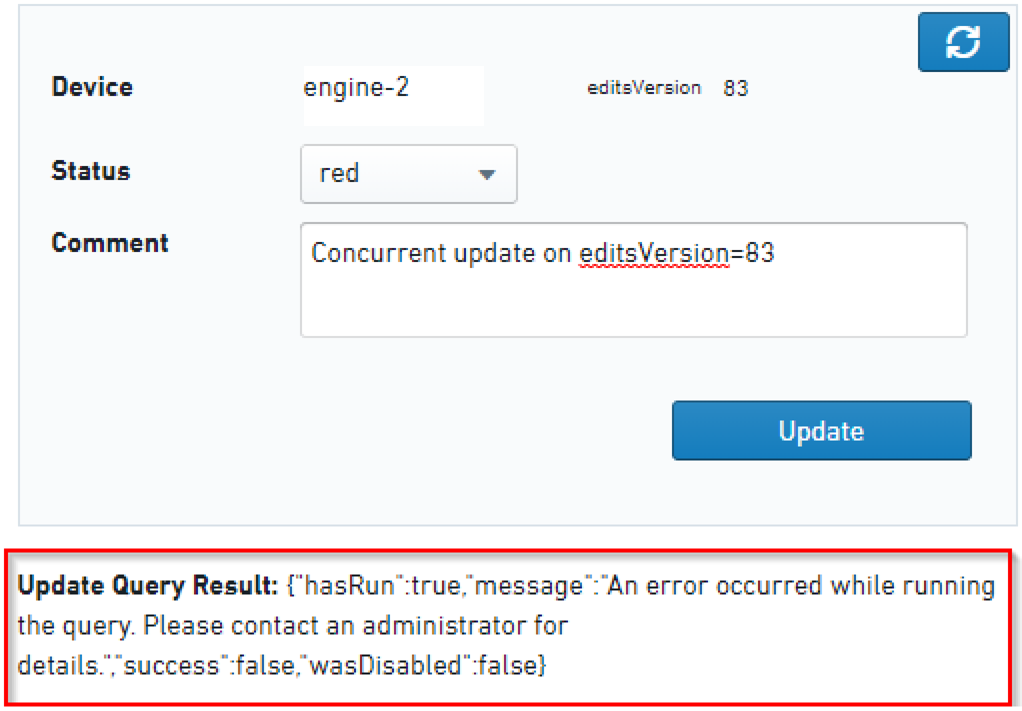
DETECTED CHANGES

	Data at read time	Update	Currently in Database
_editsVersion	82		83
status	red	red	red
comment	Update on editsVersion=81	As well an update on editsVersion=82	Update on editsVersion=82

So pretty cool. The clash be easily visualized and the user sees what happens during his edits.

Bumping you nose

So far so good, but when switching in the development window as well into the view mode, and running the test-scenario again, something unexpected happened:



Device	engine-2	editsVersion	83
Status	red		
Comment	Concurrent update on editsVersion=83		

Update Query Result: {"hasRun":true,"message":"An error occurred while running the query. Please contact an administrator for details.","success":false,"wasDisabled":false}

The type of error indication changed suddenly. Time to contact Palantir and check for the reasons: Logan, one of my sources of expertise, explained that the obfuscation of error codes in

the view mode is unfortunate for this situation, but implemented by intention, since detailed error message can have the problem of revealing unintendedly information.
The operations are still safe, but just guessing that the error was driven by a concurrent update might jeopardize the user experience.

Logans recommendation is to rather avoid the bad surprise and work on a better user guidance during editing to keep the surprise moments lower.

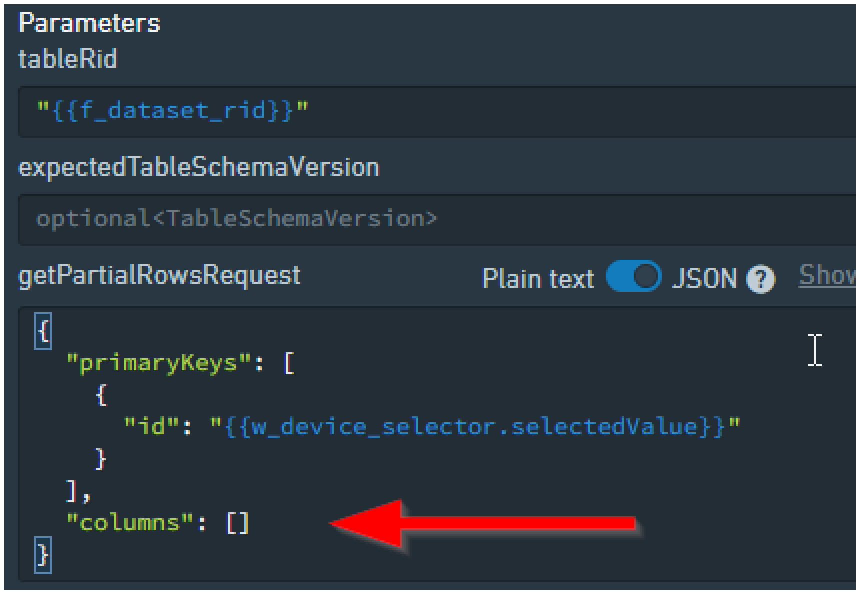
Version 2 – Indicate updates that happen during editing

A bad user experience due to a failed update can be avoided (reduced) by indicating during the editing process that someone else changed the data in between.

The needed architecture elements are available: Slate allows to implement “polling queries” that are executed in given interval. With the “get Partial Rows“- Endpoint, there is a mean to have the amount of data in the polling request as small as possible, and with the toast, there is a mean in the UI that allows to show something without interrupting the editing process.

Creation of an update check query

The update check query is basically a getRows query. The query attribute “columns” is left empty.



Parameters
tableRid

```
"{{f_dataset_rid}}"
```

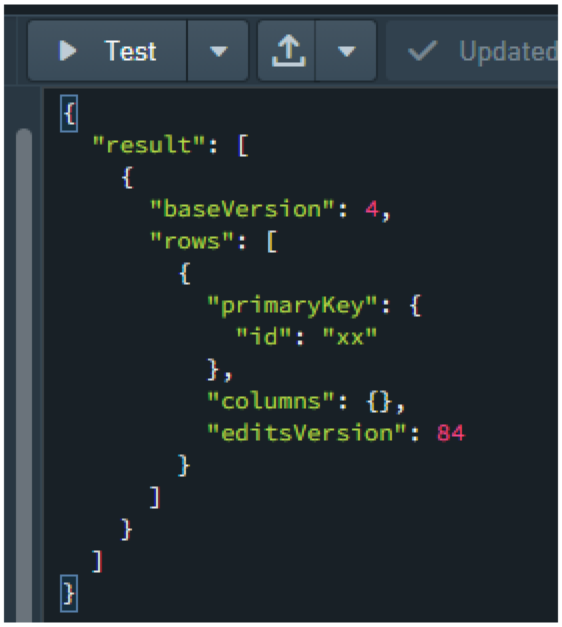
expectedTableSchemaVersion

```
optional<TableSchemaVersion>
```

getPartialRowsRequest

```
{
  "primaryKeys": [
    {
      "id": "{{w_device_selector.selectedValue}}"
    }
  ],
  "columns": []
}
```

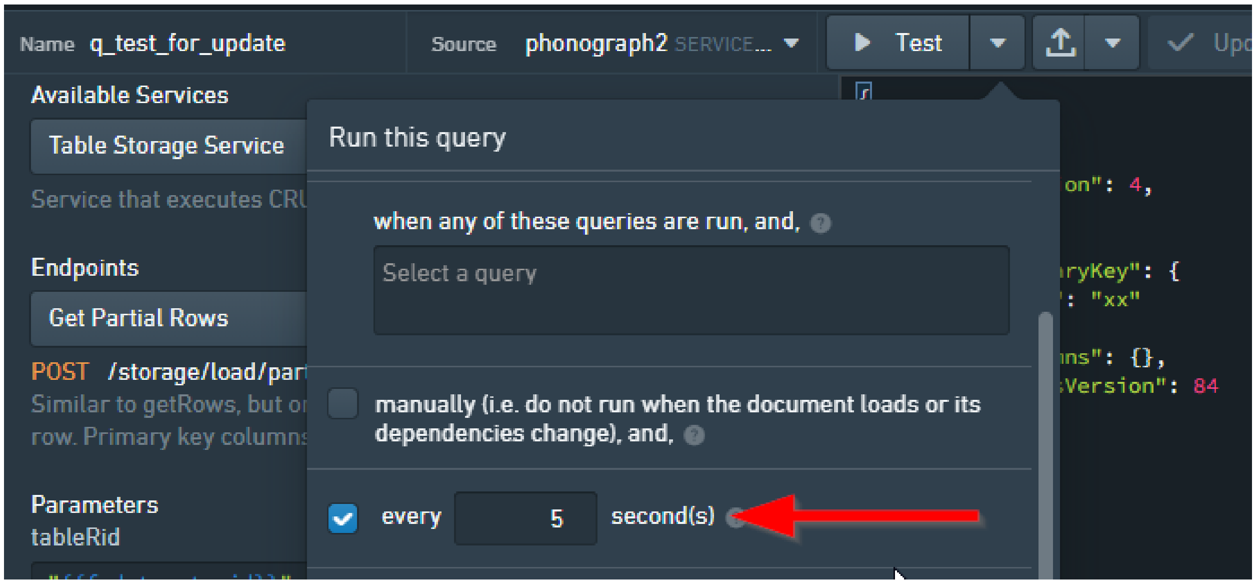
The query returns the minimum of data:



```
{
  "result": [
    {
      "baseVersion": 4,
      "rows": [
        {
          "primaryKey": {
            "id": "xx"
          },
          "columns": {},
          "editsVersion": 84
        }
      ]
    }
  ]
}
```

Using the primaryKey of the selected row, the query polls only the status of the visualized element.

Considering the small size and low complexity of the query, the polling interval can be set to a quite high frequency. In this case 5 seconds.



Name q_test_for_update

Source phonograph2 SERVICE...

Available Services

Table Storage Service

Service that executes CRU

Endpoints

Get Partial Rows

POST /storage/load/par

Similar to getRows, but o
row. Primary key column

Parameters
tableRid

Run this query

when any of these queries are run, and,

Select a query

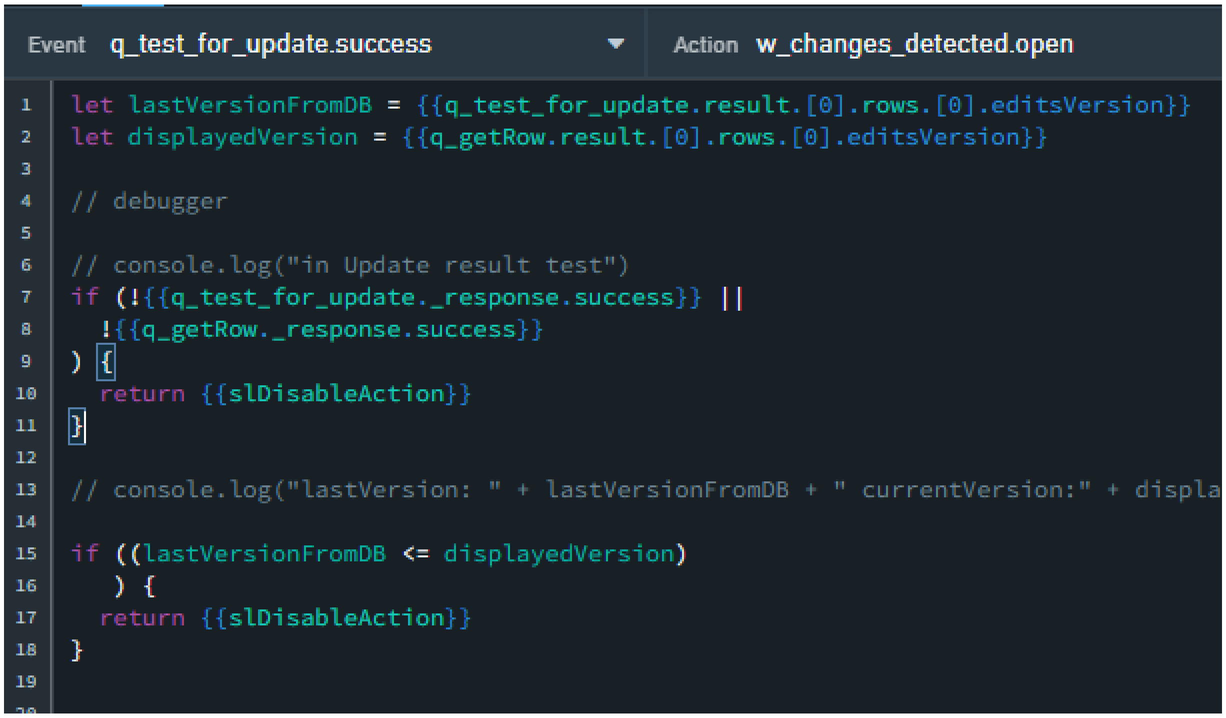
manually (i.e. do not run when the document loads or its dependencies change), and,

✓ every 5 second(s)

It is essential for the user experience, that the query for the update check is not the query, that fills the input fields after selecting the asset from the dropdown. If done like this, there will be on one hand a flickering every 5 seconds and even worse, any edit will be erased when the update takes place.

Implementation of an Update Warning

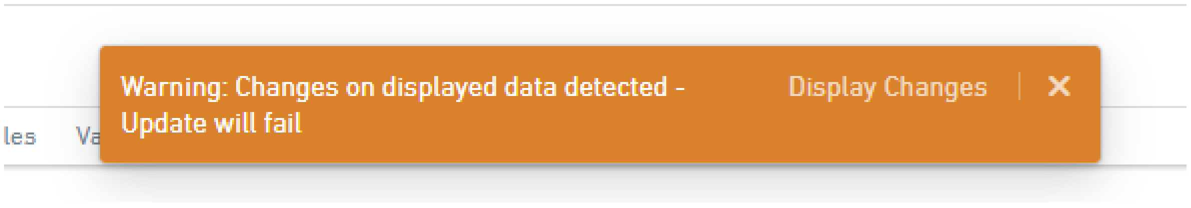
Having the editsVersion of the currently displayed row version and the result from the update check, an indication by a toast can be implemented with a proper event handler.


Event q_test_for_update.success
Action w_changes_detected.open

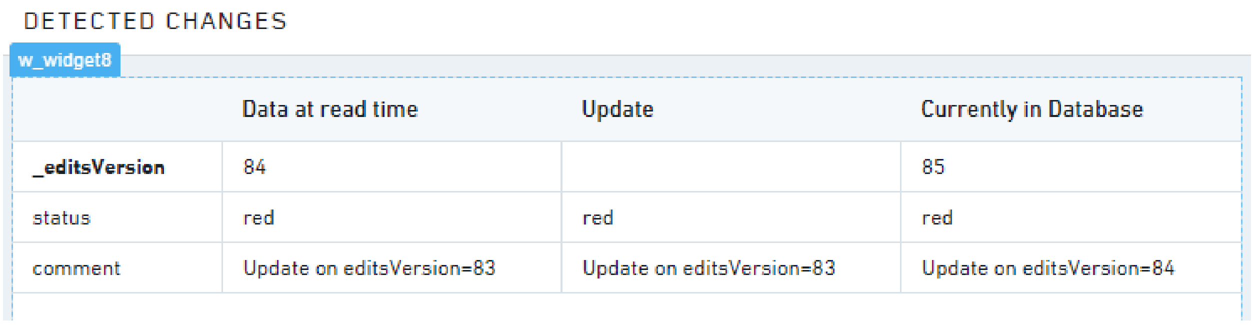
```
let lastVersionFromDB = {{q_test_for_update.result.[0].rows.[0].editsVersion}}
let displayedVersion = {{q_getRow.result.[0].rows.[0].editsVersion}}

// debugger

// console.log("in Update result test")
if (!{{q_test_for_update._response.success}} ||
  !{{q_getRow._response.success}}
) {
  return {{slDisableAction}}
}

// console.log("lastVersion: " + lastVersionFromDB + " currentVersion:" + displa

if ((lastVersionFromDB <= displayedVersion)
   ) {
  return {{slDisableAction}}
}

```

To avoid unnecessary toasts popping up at the opening of the Slate application, the response state is checked before comparing the versions.

Running a test with two opened instances of the slate application verifies the good function.


Warning: Changes on displayed data detected - Update will fail

The PopUp for the error handler indication can be easily re-used from Version 1 to show the changes.


DETECTED CHANGES

w_widget8

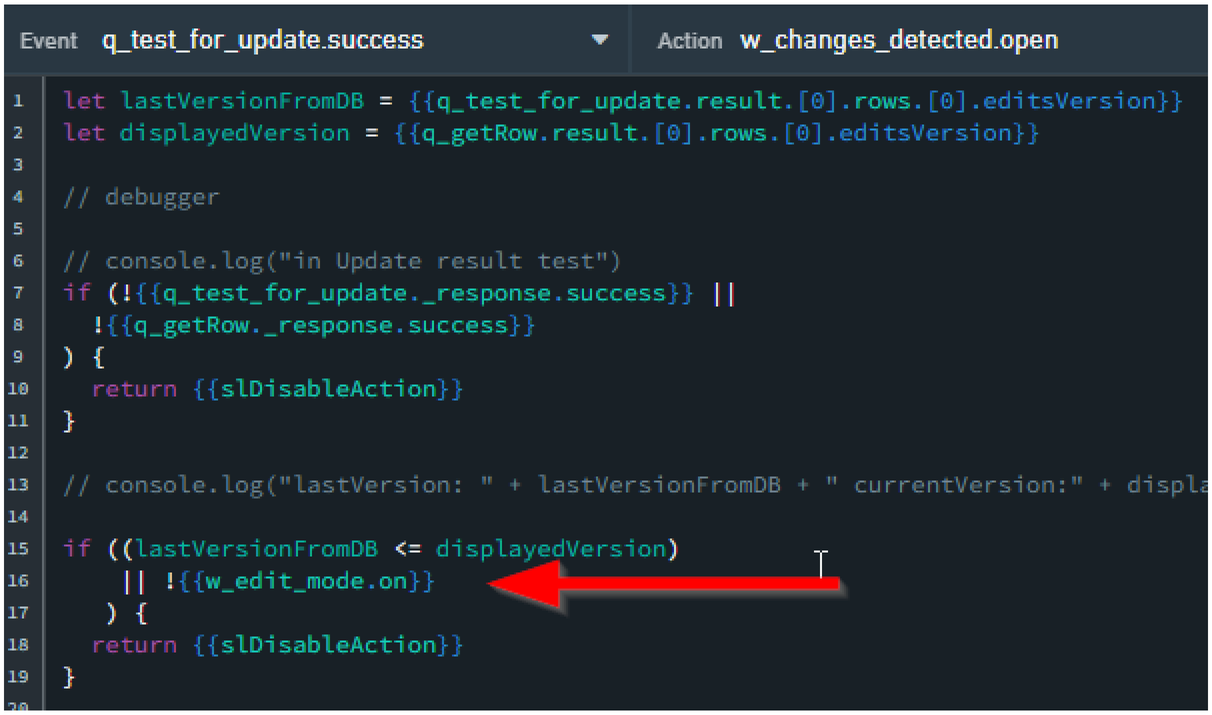
	Data at read time	Update	Currently in Database
_editsVersion	84		85
status	red	red	red
comment	Update on editsVersion=83	Update on editsVersion=83	Update on editsVersion=84

This is a strong improvement of the user experience. During visualization updates on the data are identified and displayed.

Implementation of a "view mode"

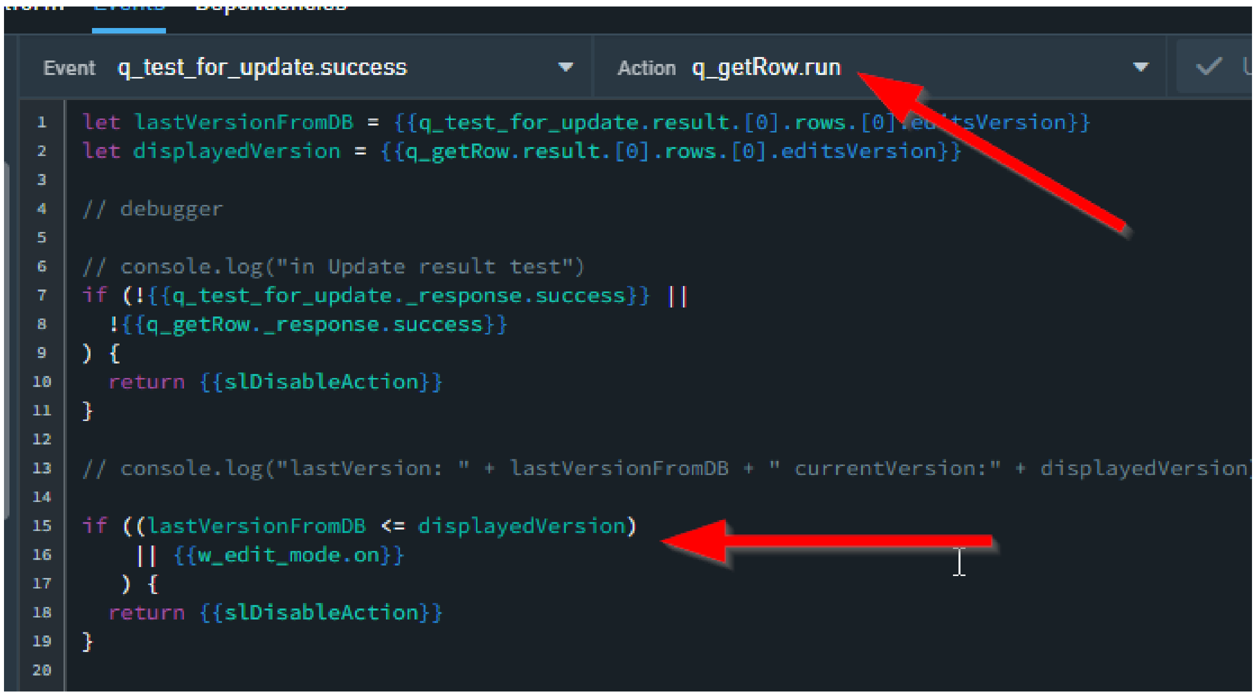
Depending on the expected update- and edit-frequency, it might be considered to implement distinction between an "edit" and "view" mode. If there is a high view- and low update rate, the users might be annoyed by the toasts. A toggle is added to the UI.

In the view-mode, the "Display change"- toast is suppressed:


Event q_test_for_update.success
Action w_changes_detected.open

```
let lastVersionFromDB = {{q_test_for_update.result.[0].rows.[0].editsVersion}}
let displayedVersion = {{q_getRow.result.[0].rows.[0].editsVersion}}

// debugger

// console.log("in Update result test")
if (!{{q_test_for_update._response.success}} ||
  !{{q_getRow._response.success}}
) {
  return {{slDisableAction}}
}

// console.log("lastVersion: " + lastVersionFromDB + " currentVersion:" + displa

if ((lastVersionFromDB <= displayedVersion)
    || !{{w_edit_mode.on}}
   ) {
  return {{slDisableAction}}
}
```

Instead of the toast, the main to get the row content query is executed (or thinking in events/actions, the run of the main query for the UI is not suppressed).



```
let lastVersionFromDB = {{q_test_for_update.result.[0].rows.[0].editsVersion}}
let displayedVersion = {{q_getRow.result.[0].rows.[0].editsVersion}}

// debugger

// console.log("in Update result test")
if (!{{q_test_for_update._response.success}} ||
  !{{q_getRow._response.success}}
) {
  return {{slDisableAction}}
}

// console.log("lastVersion: " + lastVersionFromDB + " currentVersion:" + displayedVersion

if ((lastVersionFromDB <= displayedVersion)
    || {{w_edit_mode.on}}
   ) {
  return {{slDisableAction}}
}
```

Running the main-query only in case of updates will as well improve the user-experience, since a “update flicker” of the main data display happens only in case of changes.

Solve Timing Issues

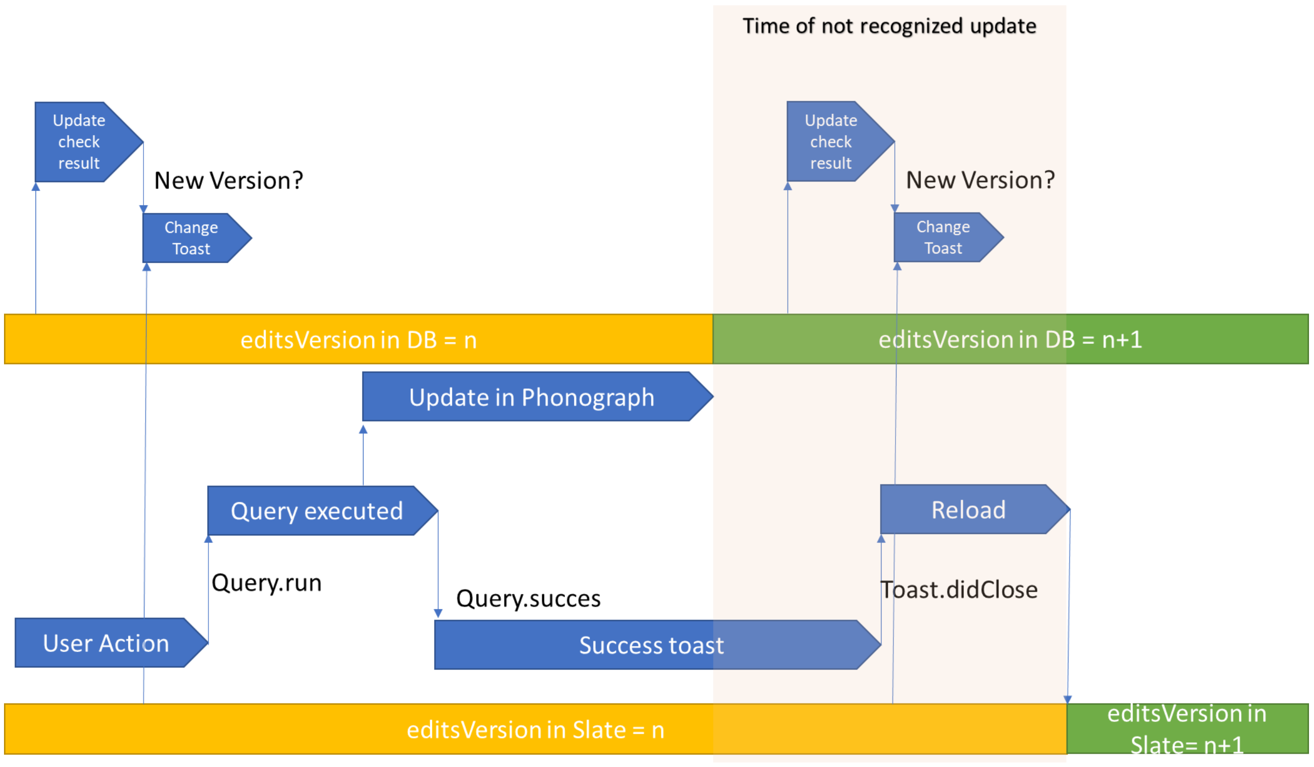
During the test it happened, that sometimes, right after the update of the row a data-update was wrongly indicated. What happened here?

The reason of this behavior is the through and through asynchronous nature of Slate and Phonograph. Every flow of control is a set of events and actions:

- A button is clicked, and sometime later, the update-query is executed.
- The query is executed and sometime later a success- or failure event can be catched.
- A failure event is catched and sometime later the toast is opened.

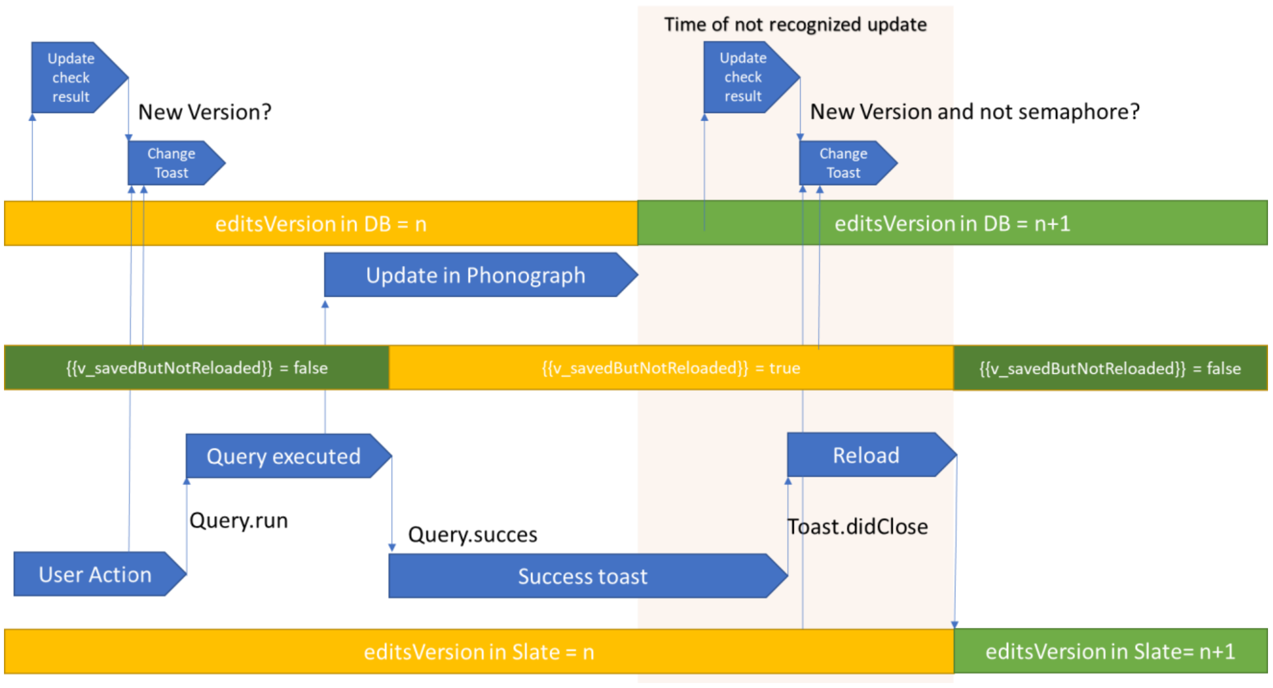
…. And so on.

With Phonograph being a non-transactional data storage another dimension of asynchronity is added: It is not guaranteed, that every read access executed after the update, returns the updated data. It can take up to seconds, until read-requests after an update will return the new data. This happened in the case the spurios update indication.



The update checker identified the update before the UI reloaded the last version. There are various patterns to cope with this. The easiest follows the old programmers rhyme. “If I have no ideas anymore, I just set up a semaphore”.

A slate variable "v_savedButNotReloaded" is introduced. When the update is launched it is true and when the final reload happened, it set back to false. During this period the update-check is suspended.



Note: Another option would be to have another update-check query that runs only when the "v_savedButNotReloaded" is true with a refresh-rate of 1s and launches automatically the reload as soon as the update is available in phonograph. A third option is to keep a front-end state inside of slate until the refresh is available, but to keep the application "simple", which is already now not the case, these options are not added.

References

Altstaedt, K. (2021). *Palantir Foundry by Use Cases.* Self published.

David Xiao, S. E. (2021). *Rethinking CRUD For REST API Designs.* Retrieved from Palantir Blog: https://blog.palantir.com/rethinking-crud-for-rest-api-designs-a2a8287dc2af