# Proskomma v0.9 Editing

by Mark Howe

May 2022

# How things work now

- **Proskomma ingests USFM or USX**. This is a relatively expensive, multi-step process during which the content is reorganized to ease querying. For example,
    - headings, footnotes etc are separated from canonical text into their own sequence
    - chapter and verse markers are turned into scopes with start and end markers
    - those markers are moved around so that, eg, a block never begins with a closing verse tag
    - non-semantic whitespace is tidied up
    - …
- **The AGHAST editor PoC uses Proskomma Render** which
    - runs a ‘kitchen sink’ query to export almost all the data as JSON
    - walks over that JSON, generating many types of callback, some of which are handled by the AGHAST code to generate JSON that the Slate editor can use
- Slate can then perform edits on the AGHAST JSON
- To write back the Proskomma, the AGHAST JSON is transformed into another JSON format that is close to the internal structure of Proskomma (but which is not close to the result of the kitchen sink query). This JSON is submitted to Proskomma via a GraphQL mutation, and used to replace the content of the existing sequence.

**This approach works, but is not ideal** because

- the process is not symmetric, ie the format of data coming out of Proskomma is different to the format going into Proskomma
- The output process uses Proskomma Render which is designed for much more complex rendering tasks, and which is relatively slow for this application
- AGHAST itself is basically a label for one way of using Slate JSON, which is not suitable for other types of editor, and which itself is proving quite hard to work with
- The write-back process is quite direct, without the multi-step checks and transformations of USFM/USX import, which means that data integrity within Proskomma depends on the editor effectively recreating those checks and transformations.

# How things will work in v0.9

## Overview

The main changes are

- **Explicit, well-defined processing layers**
- **Separation of concerns** between
    - Proskomma internal structures
    - Editor-ready JSON format
    - Stripping/merging of aligned markup such as uW word alignment
- **Symmetric read/write processes**
- **Editor-independent formats**

This should mean that

- **It is much easier to test each processing layer**, including by taking the output of each layer and feeding it straight back as input
- **Proskomma data integrity becomes a Proskomma concern**
- **Any editor technology may be used** to work with the editor-ready JSON, either directly or by converting to another format such as Slate JSON or HTML via an editor-specific process.
- **Addition/removal of markup becomes transparent to the editor** (because the base format is the same whether or not the optional markup is present)

## Proskomma Core

### *Output*

New *PERF*[1] GraphQL query field for sequences.

### *Input*

New *writePERF* GraphQL mutation that requires

- Document ID
- Sequence ID
- PERF to be written

### *Transformation*

The aim is to represent Proskomma internals in a way that is relatively easy for Proskomma to generate and consume, but which does not require the editor to manage concerns that are not present in USFM. This means

- Data is read and updated one sequence (ie one multi-paragraph flow of data) at a time.
- tokens are concatenated into a single string wherever possible, ie whenever they are not separated by markup. In simple USFM this means a single string per verse. Those strings may be edited, and will be split into Proskomma tokens using the regex that is also used by the USFM parser.
- Character-based markup (ie styling within a USFM paragraph) should be nested, rather than represented using Proskomma's milestone approach, since nested JSON is easier for most editors to work with. That nesting will be flattened on the way back into Proskomma.
- Chapter and verses should be represented by a single marker, as in USFM. The end markers, as well as verse/verse-range distinctions, will therefore be removed on export and recalculated by Proskomma on input.
- All other markup will be represented with start and end milestones. The editor will need to ensure that milestones are matched correctly, potentially across multiple paragraphs. In practice, this is the kind of markup that will probably be stripped out before the content reaches the editor.
- On input, Proskomma may also garbage-collect sequences for the edited document, in case, eg, a heading or footnote has been deleted.

Proskomma will validate PERF input using a JSON schema. It will also check that sequences referenced via grafts exist.

1 Proskomma Editor-Ready Format

## MkPERF

MkPERF is a JS class that maintains a PERF representation of a sequence. It provides

- *createBlock()* to create and populate a new block
- *readSequence() to* access the entire sequence
- *updateBlock()* to modify one block
- *updateSequence()* to modify the entire sequence
- *deleteBlock()* to remove a block
- *fetchFromPk()* to refresh MkPERF state from Proskomma
- *submitToPk()* to rewrite the sequence in Proskomma

All methods return

- new, possible unmodified PERF which the editor should render, to avoid any skew due to artefacts of the multilayered processing
- an array of error messages

### *Constructor*

- Proskomma Instance
- Document ID (string)
- Sequence ID (string)
- StrippableScopes (Array of strings using *startsWith()* logic)
- allowOutOfOrderVerses (boolean)
- allowMissingVerses (boolean)
- allowInvalidPERF (boolean)

The constructor performs an initial *fetchFromPk()*.

### *createBlock()*

- New Block Number (integer)
- PERF for new block (object)

### *readSequence()*

### *updateBlock()*

- Block Number (integer)
- PERF for block (object)

### *updateSequence()*

- PERF for sequence (object)

### *deleteBlock()*

- Block Number (integer)

### *fetchFromPk()*

### *submitToPk()*

- PERF for the sequence (object)

### *Stripping and merging markup*

When enabled, specified scopes will be removed as part of the read and merged as part of the write. The scopes are specified as the start of the scope string, as elsewhere in Proskomma.

**Internal format**

For start scopes:

- block no
- payload of token following scope
- occurrence count of that token in block
- total number of occurrences of that token in block

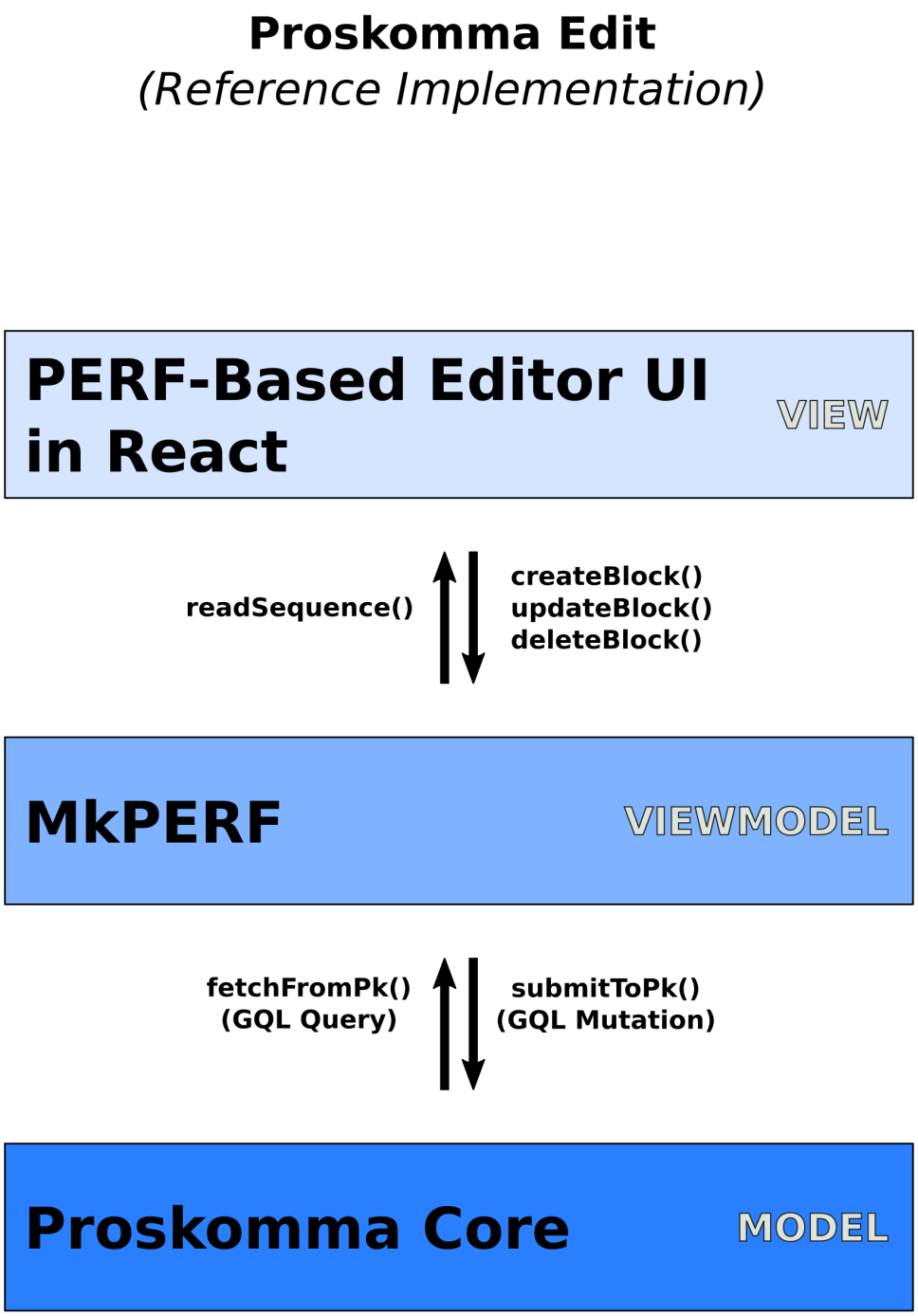
For end scopes:

- block no
- payload of token preceding scope
- occurrence count of that token in block
- total number of occurrences of that token in block

## Editors

### *Proskomma Edit*

Proskomma Edit is the reference implementation of a PERF-based Scripture editor. It uses PERF directly without any intermediate format, and without any third-party rich-text editor library. The emphasis will be on demonstrating that the PERF layers work rather than on a superlative UX.



### *Third Party Editors*

Other editors may work directly with PERF, or may convert PERF into another form of JSON, or into another data format such as HTML or XML. They may use MkPERF as a cache that may be edited one block at a time, or as a way to work directly with Proskomma.

Developers of 3rd-party editors may elect to subclass MkPERF, adding new, editor-specific methods, eg *readHTML()*.