proskomma

Release 0.6

Mark Howe

May 13, 2022

CONTENTS

CHAPTER

ONE

A SCRIPTURE RUNTIME ENGINE

1.1 Background

1.1.1 The Challenges

Why is USFM so popular?

USFM and its XML sibling USX have become the format of choice for Bible translation. There are several good reasons for this:

- USFM is a standardized dialect of Standard Formatting Markers (SFM) that have been used by many organizations for decades. (SFM itself was initially used mostly for dictionaries before people starting forcing Scripture into the format, and then the early SFM Bibles were often edited by the Shoebox editor designed for dictionaries, replacing shoeboxes full of dictionary cards).
- The backslash-based markup, reminiscent of a 1980s word processor, has proved to be easy for Bible translators to use - at least for the most common tasks - including those with limited previous experience with computers
- A rich technological ecosystem has been created around USFM, including editors such as Paratext and data stores such as the Paratext Registry and the Digital Bible Library
- USFM is now at the basis of many Bible translation training programs, and almost all Bible translation communities have local experts in this format

In addition, other formats such as OSIS have failed to replace USFM in the past. Given the number of translation projects now using USFM, and the close deadlines of projects such as Vision 2025, it seems likely that USFM will continue to be the translation format of choice for the foreseeable future.

Why is USFM so hard to process?

Evolving Needs

USFM was defined at a time when, typically, the only output format was the printed page. Each new edition was a major event, prepared by many people including typesetters and professional printers, over an extended period.

In 2020, Scripture is published in many different digital formats. Even printing has changed radically with the advent of desktop publishing and print on demand. Modern translation workflow encourages rapid, incremental publishing as part of the translation process. USFM was not designed with any of these needs in mind.

For example, in many digital applications, queries like *"What is in chapter 1?"* or *"Give me John 3:4-4:2"* are common and vital. Such queries are a non-issue for traditional typesetting, where a new chapter is just another item to render on a page (maybe implementing widow/orphan logic). To put it another way, the reader is the search engine for a book.

Algorithmic chapter and verse selection is therefore hard in USFM. It becomes much harder once retrofitted features such as cp are added into the mix.

Getting Past the Markup

Backslash-based markup was quite common in 1980. Today many programmers meet it for the first time in USFM. Modern technologies such as XML and JSON are of no help. A character-based parser is required, and these remain hard to write - or at least to write well. The vagueness of parts of the USFM specifications does not help here. (See, for example, the notes on whitespace.) Neither does implicit marker closing or the workaround required to avoid it.

The semantics of USFM can also be hard to unpick. For example, sometimes a paragraph is literally a paragraph of canonical content, such as p. Sometimes a paragraph is a non-canonical heading (not to be confused with a canonical heading). Sometimes a paragraph is a chapter marker (not to be confused with a published chapter). Sometimes a paragraph is more like a processing instruction, such as cl which, in addition, can mean two different things depending where it is used.

Character-level markup is a similarly mixed bag, including inline footnotes and cross-references. Inline hypertext in itself is not unusual - it's a feature of many W3C standards. Such features become more challenging in USFM because of Bible BCV (see below).

USFM 2 includes introductions, which use tags with different names that are often equivalent to tags used elsewhere. USFM 3 introduced an optional "end introduction" marker (which, in practical terms, is useless since that marker may or may not appear in any particular valid USFM 3 document). Features such as sidebars have been shoehorned into the existing structure, and it seems like every type of peripheral content is now delimited a different way.

Chapter and Verse

Modern translations are structured in terms of semantically meaningful divisions such as sections, paragraphs and sentences. However, the vast majority of Christians and church traditions expect content to be rendered in terms of **Book, Chapter, Verse** (BCV). There are myriad variations on the details of BCV across traditions and languages and, in many cases, those divisions cut across the more modern structure hierarchy. It is therefore decidedly non-trivial to find the text within a BCV range while preserving other features of the translation, all in a format as familiar to younger programmers as data storage on audio cassettes.

Partial Solutions

USFM obviously gets processed, a lot. But, because of the challenges involved, solutions tend to be partial and fragile. The easy things get done quickly, progress beyond a certain point becomes exponentially harder, and at some point progress stops because the result is "good enough", at least with the limited range of texts that need to be processed. The resulting code is fragile, especially when confronted with texts that use a different subset of USFM features, and at this point the easiest thing is to start writing a different, partial, fragile solution for the new use case.

The difficulty of processing USFM therefore ends up creating a barrier to using Scripture in modern media. It is very hard to spin this as A Good Thing.

USX

USX is the XML expression of USFM. It has advantages over USFM, including

- processing using standard XML tools
- a formal schema for validating USX documents
- explicit, XML representation of the implicit markup-termination rules of USFM

However,

- USX does not address any of the deeper semantic issues inherited from USFM. It cannot address them without breaking roundtripability with USFM.
- USX is not suitable for editing by hand, and cannot therefore be exposed to the end user in an editor.
- In the XML world, there can be no such thing as "invalid USX". If something doesn't validate according to the schema, it isn't USX. This may be because of one misused tag, or because the "invalid USX" is actually some completely unrelated XML vocabulary such as an SVG illustration. This is a problem if less-than-perfect translation projects need to be serialized.
- The flat structure of USX proves challenging to process using standard XML tools. (A paper presented at XML Prague 2011 explored some of these challenges with respect to XSLT.)
- USX has some design quirks that appear to have been driven by conciseness rather than convenient parsing. For example, published chapter number can be an attribute or character markup.

USFM/USX v3

v3 of USFM includes many extensions. Some of these, such as chapter and verse milestones are intended to address the *"What is in chapter 1?"* above. Others add support for new features such as extended study content.

Unfortunately, the addition of chapter/verse milestones makes USX generation much harder, and makes manual editing of USX almost impossible. This is because chapter/verse information has been denormalized into multiple locations that must be maintained in parallel. None of this helps with "published chapter/verse" markup. In addition, the precise tag order for these milestones is under-defined so, in practice, the way Paratext does it becomes the *de facto* standard.

In addition, some USFM 3 features look more like Paratext-specific features. See, for example, the @srcloc word alignment feature which does not map onto the word alignment of many major Bible translation ecosystems.

Towards a Generic Solution

> **Easy things should be easy, and hard things should be possible.**
> *Larry Wall*
> Creator of Perl and one-time Bible translation intern

It follows from the description above that a number of approaches are not viable:

- USFM is very unlikely to be replaced as a translation format because it is widely used, and because previous attempts to replace it have been expensive failures.
- Attempts to "fix" USFM/USX through extensions may address some specific pain points, but at the cost of making the overall processing model more and more byzantine.
- Simplifying USFM would be very hard because, while almost no-one uses every single feature, every feature is used by someone.

Proskomma assumes that ranking texts will continue to be stored in USFM or USX. It attempts to provide an explicit processing model that

- represents USFM concepts using a small number of consistent building blocks
- addresses some of the semantic pain points
- is flexible enough to handle a range of use cases.

Note: One consequence of this approach is that USFM is unlikely to be fully roundtripable. It should be possible to export USFM from Proskomma, but that USFM may not be a character-for-character match with the imported USFM.

1.1.2 Proskomma Features

USFM and USX Parsing

Proskomma provides multiple lexers that perform the first step of document import. The USFM and USX lexers are intended to produce an identical internal representation of the document, which means that a single toolchain can handle the rest of the processing.

Note: One consequence of this is that some USX features are discarded during parsing. This is notably the case of *ref/@loc* which, in USX provides BCV references in a standard format. This information cannot sanely be maintained by a manual editor and, in any case, the information is not available in USFM. Rather than relying on one closed-source editor to generate this information, the Bible-publishing world needs a standard way to share reference parsing information.

Multiple Content Types

Scripture has long been at the centre of other documents such as concordances, lexicons, commentaries and translation notes. Rather than shoehorning this content into USFM, Proskomma supports multiple document types, based on the same model, but with domain-specific markers. This approach facilitates processing across multiple content types.

Proskomma also supports multiple representations of the same input format. In particular, it is possible to represent Scripture per-paragraph (as USFM does) or per-chapter/verse (as many applications assume) or both. This delivers faster results and less tortured application code.

A Simple, Consistent API Model

USFM includes paragraph-level markup, character-level markup, word-level markup, milestones, attributes, chapters, verses… Proskomma represents all that using three building blocks:

- *tokens* (text)
- *scopes* (markup around a portion of text)
- *grafts* (something to potentially insert at a specific point in the text)

Grafts are used to split Scripture into *sequences*. The main sequence contains canonical Scripture. Introductions, title pages, headings, sidebars, footnotes, cross-references and other non-canonical data is placed in separate sequences, linked to the parent sequence. The advantage of this approach is that each sequence represents exactly one thing, that you never need to skip past text, and that it is easy to include related material explicitly when necessary.

Efficient Storage

Modern computing favours tree representations such as XML and JSON. Such representations allow for convenient, flexible processing. However, they also tend to take up a lot of memory because of 64-bit pointers. In-memory XML often becomes between 10 and 20 times larger when represented as a DOM tree.

The Proskomma building blocks are tiny, which would result in lots of pointers and huge in-memory document sizes. Proskomma therefore makes extensive use of succinct data structures:

> A succinct data structure is a data structure which uses an amount of space that is "close" to the information-theoretic lower bound, but (unlike other compressed representations) still allows for efficient query operations… Unlike general lossless data compression algorithms, succinct data structures retain the ability to use them in-place, without decompressing them first.

Flexible Queries

Proskomma uses GraphQL as the basis of its API. Queries may be tailored to provide just the information that is needed for any particular task, structured in a way that is convenient to process.

1.2 Getting Started

1.2.1 Installing Proskomma

Proskomma is a Javascript library. The simplest way to install it is using npm:

```
npm install proskomma
# or
yarn add proskomma
```

The proskomma module returns an object you will probably want to dereference:

```
const { Proskomma } = require('proskomma');
// or
import { Proskomma } from 'proskomma';
```

1.2.2 Interacting with Proskomma

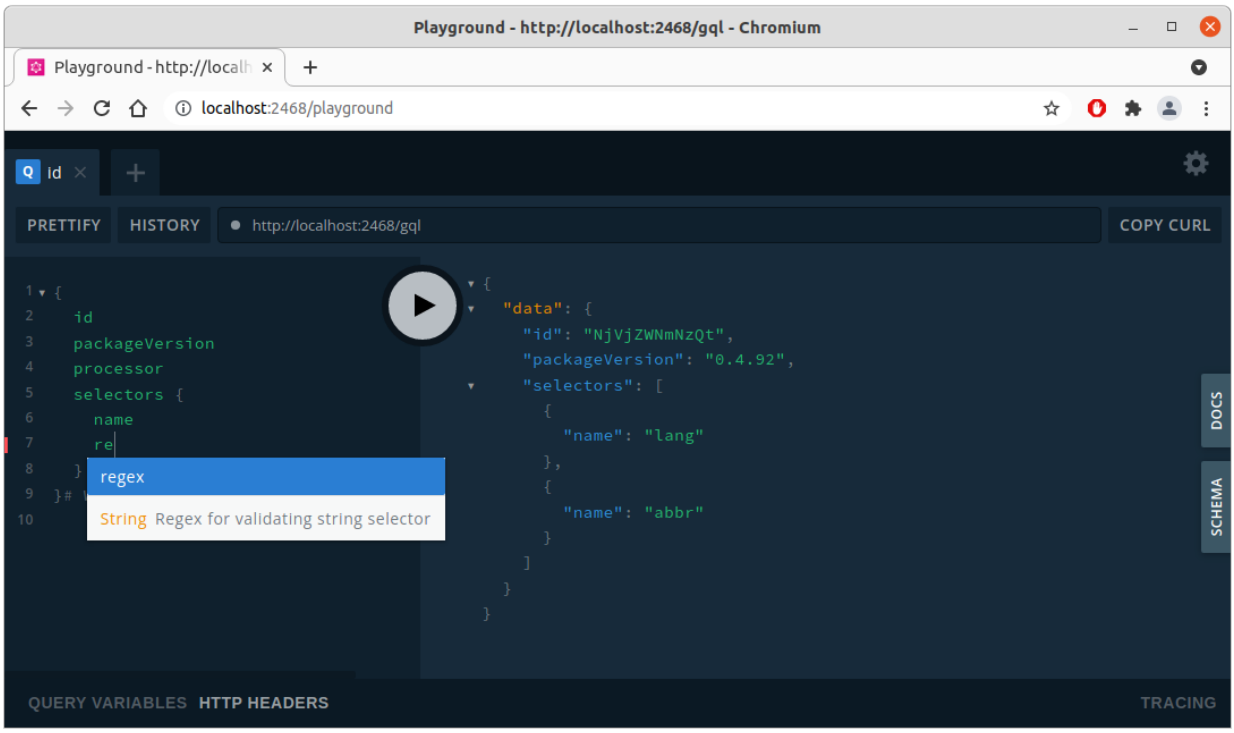
The officially-supported interface for almost all Proskomma operations is GraphQL. There are multiple ways to interact with the GraphQl interface, all of which should produce the same results.

In several of the following examples, the query is

```
{ id }
```

This is the shortest possible Proskomma query, and returns the unique id of the Proskomma instance. (While short, this query is not that useful, except for 'hello world' testing, and for checking that the Proskomma instance has not been overwritten unintentionally.)

Interacting with Proskomma using a GraphQL Playground



Fire up proskomma-node-express:

```
git clone git@github.com:mvahowe/proskomma-node-express.git
cd proskomma-node-express
npm install
npm run dev
```

Node Express should now be listening on port 2468 of localhost. The playground is at

```
http://localhost:2468/playground
```

Interacting with Proskomma using a Node script

There are simple scripts for using Proskomma from the command line in the Proskomma repo. eg

```
cd scripts
node ./do_graph.js ../test/test_data/usfm/hello.usfm example_query.txt
```

The script imports a USFM or USX file into Proskomma and then runs the query contained in a text file against it, outputting the result to *stdout*. on un*x systems you can redirect that result to a file:

```
node ./do_graph.js ../test/test_data/usfm/hello.usfm example_query.txt > ~/my_result.json
```

If you want to write your own script to do something different, the code for *do_graph.js* would be a good starting point.

Interacting with Proskomma from Javascript

The typical, basis workflow is

- import Proskomma
- instantiate Proskomma
- import some data
- query Proskomma (async)

```
const fse = require('fs-extra');
const { Proskomma } = require('proskomma'); // Import Proskomma

const pk = new Proskomma(); // Instantiate Proskomma

const content = fse.readFileSync('my_usfm_file.usfm');
pk.importDocument( // Import Data
   {
      lang: 'en',
      abbr: 'kjv',
   },
   'usfm',
   content,
);

const query = '{id}';
pk.gqlQuery(query) // Query Proskomma
  .then(output => console.log(JSON.stringify(output, null, 2)))
  .catch(err => console.log(`ERROR: Could not run query: '${err}'`));
```

Interacting with Proskomma using Curl

To use an HTTP client with Proskomma, you first need an HTTP server. (Unlike most systems offering a GraphQL interface, Proskomma does not itself wait on a socket.) So the first step is to install and fire up proskomma-node-express:

```
git clone git@github.com:mvahowe/proskomma-node-express.git
cd proskomma-node-express
npm install
npm run dev
```

Node Express should now be listening on port 2468 of localhost. You can test this by pointing your web browser at

```
http://localhost:2468
```

Now open a second terminal window (since the server needs to continue running) and type

```
curl -X POST http://localhost:2468/gql -d 'query={ id }'
```

Interacting with Proskomma using the Chaliki Electron App

Chaliki is an electron app built around Proskomma. It comes preloaded with several complete translations and provides a desktop UI.

```
git clone git@github.com:Proskomma/chaliki.git
cd chaliki/data
# unzip translations.zip into the data/ directory (ie 10 files directly under data/)
yarn install
yarn start
```

The menu, top left, should be fairly self-explanatory. Each page demonstrates the use of a different query to meet a different use case. You can see the query, and then modify it yourself, by clicking the *Inspect Query* button at the top of each page.

1.2.3 Tutorials

Hello Proskomma Tutorial

This tutorial shows how to get up and running with Proskomma, in node. We will import and query a short sample document. All the code and data for the tutorial is presented inline. You will need to copy and paste the code into a file called `tutorial.js`, and the USFM into a file called `psa.usfm`. You will also need to be able to type commands via a terminal emulator such as Bash.

Installing and Importing Proskomma

Create a directory for this project and move into it:

```
mkdir my_proskomma_tutorial
cd my_proskomma_tutorial
```

Install proskomma:

```
npm install proskomma
```

(If this command throws an error, check that you have already installed node and npm on your system, and that you have access to the Internet.)

Create a file called `tutorial.js` in your project directory, open it with the text editor of your choice (which is probably Emacs) and enter the following code:

```
const { Proskomma } = require('proskomma');
```

Save that file and, from your terminal emulator, type

```
node tutorial.js
```

This command should run without throwing an error and without producing any output. You could add

```
console.log(Proskomma);
```

to convince yourself that the module has been loaded.

Interacting with Proskomma

The next step is to make an instance of Proskomma:

```
const pk = new Proskomma();
```

Add this line to your code file, so that the whole file looks like this:

```
const { Proskomma } = require('proskomma');
const pk = new Proskomma();
```

You could peek at Proskomma internals by adding

```
console.log(pk);
```

You could get, say, the unique id of this Proskomma instance:

```
console.log(pk.processorId);
```

However, most Proskomma internals are hard to access in this way, because they are stored in a succinct, binary format. In almost all cases it is better to interact with Proskomma via GraphQL. The GraphQL equivalent to the previous example is

```
{ id }
```

This looks a lot like a javascript destructuring assignment, such as the first line of our code file. It can be read as *return the field called 'id' at the top level of the GraphQL structure*. Most GraphQL queries are longer (and more useful) than this.

GraphQL queries are asynchronous, which means that they return a promise. You can see this by changing your script to

```
const { Proskomma } = require('proskomma');
const pk = new Proskomma();
console.log(pk.gqlQuery('{ id }'));
```

For the sake of this tutorial, we'll create a simple, asynchronous helper function that waits for the promise to be resolved and then prints the result:

```
const { Proskomma } = require('proskomma');
const pk = new Proskomma();

const queryPk = async function (pk, query) {
    const result = await pk.gqlQuery(query);
    console.log(JSON.stringify(result, null, 2));
}

queryPk(pk, '{ id }');
```

You should see the following output:

```
{
  "data": {
    "id": "NGIxMTM0N2It"
  }
}
```

The GraphQL result is an object that contains at least one of

- *data*: the requested information, as nested objects and arrays
- *errors*: any issues with the query as a whole, or with specific fields within the query

The nested structure of *data* corresponds to the structure of the query. If we were to change the query to

```
queryPk(pk, '{ idx }'); // No such field!
```

we would see the following output:

```
{
  "errors": [
    {
      "message": "Cannot query field \"idx\" on type \"Query\". Did you mean \"id\"?",
      "locations": [
        {
          "line": 1,
          "column": 3
        }
      ]
    }
  ]
}
```

Our Proskomma instance does not yet contain any data. If we ask for an array of documents

```
queryPk(pk, '{ documents { id } }');
```

the array will be empty:

```
{
  "data": {
    "documents": []
  }
}
```

Importing Scripture

Proskomma imports content from Javascript strings. In this case we will read that string from a file, but we could also download content from an API, or read it from a Javascript module, or even include the content inline in the script. Create a file called `psa.usfm`, using your text editor, paste the following text into it, and save it.

```
\id PSA unfoldingWord® Simplified Text (truncated)
\ide UTF-8
\toc1 The Book of Psalms
\mt1 Psalms
\c 150
\q1
\v 1 Praise Yahweh!
\q1 Praise God in his temple!
\q2 Praise him who is in his fortress in heaven!
\q1
```

(continues on next page)

(continued from previous page)

```
\v 2 Praise him for the mighty deeds that he has performed;
\q2 praise him because he is very great!

\q1
\v 3 Praise him by blowing trumpets loudly;
\q2 praise him by playing harps and small stringed instruments!
\q1
\v 4 Praise him by beating drums and by dancing.
\q2 Praise him by playing stringed instruments and by playing flutes!
\q1
\v 5 Praise him by clashing cymbals;
\q2 praise him by clashing very loud cymbals!

\q1
\v 6 I want all living creatures to praise Yahweh!
\q1 Praise Yahweh!
```

This text represents part of the Psalms, from a translation by Unfolding Word, in a format called USFM. USFM is widely used by Bible translators, and is one of the formats that Proskomma can import.

To read this text file, we will use *fs-extra* and *path*, the popular Node modules for interacting with filesystems:

```
npm install path
npm install fs-extra
```

Content may be imported via methods, or via GraphQL mutations. We're going to take the GraphQL mutation route. Modify your script as follows:

```
const path = require('path');
const fse = require('fs-extra');
const { Proskomma } = require('proskomma');
const pk = new Proskomma();
let content = fse.readFileSync(path.resolve(__dirname, './psa.usfm')).toString();

const queryPk = async function (pk, query) {
    const result = await pk.gqlQuery(query);
    console.log(JSON.stringify(result, null, 2));
 }

 const mutation = `mutation { addDocument(` +
     `selectors: [{key: "lang", value: "eng"}, {key: "abbr", value: "ust"}], ` +
     `contentType: "usfm", ` +
     `content: """${content}""") }`;

 queryPk(pk, mutation);
```

A mutation is a GraphQL operation that modifies the internal state of the model behind the graph (ie Proskomma in this case). The mutation type is called *addDocument*. It takes three arguments:

- *selectors*: an array of key-value pairs that, together, describe the collection (docSet) to which the document will be added
- *contentType*: the format of the input - USFM in this case
- *content*: the string containing the content, which is triple-quoted so that quotes within the USFM do not mess

up the GraphQL.

The output is:

```
{
  "data": {
    "addDocument": true
  }
}
```

which tells us that the *addDocument* mutation succeeded. Now let's add a query to explore the document we just imported:

```
const path = require('path');
const fse = require('fs-extra');
const { Proskomma } = require('proskomma');
const pk = new Proskomma();
let content = fse.readFileSync(path.resolve(__dirname, './psa.usfm')).toString();

const queryPk = async function (pk, query) {
    const result = await pk.gqlQuery(query);
    console.log(JSON.stringify(result, null, 2));
 }

 const mutation = `mutation { addDocument(` +
     `selectors: [{key: "lang", value: "eng"}, {key: "abbr", value: "ust"}], ` +
     `contentType: "usfm", ` +
     `content: """${content}""") }`;

 queryPk(pk, mutation);

 const dataQuery = `{ documents { id } }`;
 queryPk(pk, dataQuery);
```

This is the query we tried above when Proskomma was empty. (For the rest of the tutorial we will change the value of *dataQuery* to try different queries.) The output is now

```
{
  "data": {
    "addDocument": true
  }
}
{
  "data": {
    "documents": [
      {
        "id": "ODA4ZDdhNjgt"
      }
    ]
  }
}
```

(From now on we will not show the result of the mutation.) The *documents* array now contains one object which in turn contains the requested id.

Querying Basics

Proskomma has many, many fields, and - thanks to the power of GraphQL - those fields may be combined in many, many ways. However, all GraphQL queries are structured in a similar way. (This consistency, which GraphQL imposes, is one of the advantages of a GraphQL interface.)

We have already seen that the result is a nested object corresponding to the query. To request additional fields, you simply list them at the appropriate level in the query. So

```
{
   id
   packageVersion
   documents {
      id
      headers {
         key
         value
      }
   }
}

==>

{
   "id": "ZDAzZGQyZGYt",
   "packageVersion": "0.4.78",
   "documents": [
        {
           "id": "M2NlYmJjNDAt",
           "headers": [
              {
                 "key": "id",
                 "value": "PSA unfoldingWord® Simplified Text (truncated)"
              },
              {
                 "key": "bookCode",
                 "value": "PSA"
              },
              {
                 "key": "ide",
                 "value": "UTF-8"
              },
              {
                 "key": "toc",
                 "value": "The Book of Psalms"
              }
           ]
        }
   ]
}
```

id and *packageVersion* are at the top level of the query and thus refer to the Proskomma processor itself. *id* within *documents* refers to each document in the processor. *headers* describes metadata for each document, and each element of that array describes *key* and *value* for one header.

GraphQL queries tend to become quite deeply nested. This has the advantage of making the structure very clear. However, it often makes sense to post-process the raw result to make the data easier to use from Javascript. For example, the *headers* object could be represented as a simple object.

Fields may take arguments, typically to filter the results. For example, rather than returning all the headers as an array, we could ask for one specific header:

```
{
   documents {
      header(id:"toc")
   }
}

==>

{
   "documents": [
      {
        "header": "The Book of Psalms"
      }
   ]
}
```

The *header* field requires one argument, *id*. It returns a single string, which means that, unlike *headers*, there is no need to destructure its value. This is quite convenient… until you want multiple values from the same field:

```
{
   documents {
      header(id:"toc")
      header(id:"ide")
   }
}

==>

{
  "errors": [
    {
      "message": "Fields \"header\" conflict because they have differing arguments. Use␣
↪different aliases on the fields to fetch both if this was intentional.",
      "locations": [
        {
          "line": 1,
          "column": 15
        },
        {
          "line": 1,
          "column": 32
        }
      ]
    }
  ]
}
```

The error message in this case is quite informative. The solution is to explicitly assign labels (aliases) to the value of

each use of the field:

```
{
   documents {
      title: header(id:"toc")
      encoding: header(id:"ide")
   }
}

==>

{
   "documents": [
      {
         "title": "The Book of Psalms",
         "encoding": "UTF-8"
      }
   ]
}
```

In addition to solving the name conflict, the use of aliases here makes the result more self-documenting.

Querying Scripture

Proskomma stores Scripture as a deep hierarchy, which allows for quite sophisticated queries. There are also convenience fields to get basic information easily.

To get all the text for a document:

```
{
   documents {
      mainText
   }
}
```

To get all the text as an array of strings (one per paragraph):

```
{
   documents {
      mainBlocksText
   }
}
```

To get the book code of the document and the type of each paragraph too:

```
{
   documents {
     bookCode: header(id:"bookCode")
      mainBlocks {
         bs { payload } text
      }
   }
}
```

To get one verse:

```
{
   documents {
      cv(chapterVerses:"150:3") {
         text
      }
   }
}
```

To get several verses:

```
{
   documents {
      cv(chapterVerses:"150:3-4") {
         text
      }
   }
}
```

To get a chapter, split by verse, with the verseRange for each verse:

```
{
   documents {
      cvIndex(chapter:150) {
         verses {
            verse {
               verseRange
               text
            }
         }
      }
   }
}
```

See elsewhere in the documentation for more possibilities.

1.3 GraphQL Reference

This section introduces and links to various parts of the schema documentation that has been auto-generated using graphdoc.

1.3.1 Overall Design

Proskomma's GraphQL schema is quite deeply nested. Where many schema provide a set of top-level accessors that take a large number of arguments, and which return quasi-tabular results, Proskomma exposes a tree view, with filtering at each level. Convenience fields have been added to make the simple things simpler, but the full tree representation makes the hard things possible. Conceptually, Proskomma proceeds as if it was cascading Javascript map and filter operations to destructure a complex object. (In some cases this is exactly what happens under the hood, although, in many cases, the actual mechanism involves operations on byte arrays.)

In many places, fields are provided to return counts, eg *nDocSets*, *nDocuments*. These are often much cheaper than constructing a nested array just to count the number of elements.

Several types include *tags*. These can be used to support business logic such as workflow and translation status.

Some types such as *DocSet* and *Document* appear at a specific level in the GraphQL schema. Others, such as *ItemGroup* are used as semi-generic containers in multiple contexts. This makes some results a little messier, but also reduces the number of types that need to be understood in order to use Proskomma.

By the same logic, the *item* type is used for tokens, scopes and grafts, with three fields of the same name in all cases. Initially, Proskomma used 3 different GraphQL types, which allowed more descriptive field names. However, it was also necessary to cast items whenever they were used, which resulted in a lot of repetitive boilerplate in queries. The current approach trades a little self-documentation for considerably shorter queries.

1.3.2 Major Schema Types

Top-Level Query Fields

DocSets

Fields exist to access a specific docSet, or an array of docSets, filtered in various ways, and to get the number of docSets cheaply.

Documents

Fields exist to access a specific document, or an array of documents - filtered in various ways - and to get the number of documents cheaply.

Processor Information

Fields exist to return the id of the Proskomma instance and the package version number.

See the full schema for top-level query elements for more details.

DocSets

A docSet is a collection of documents. DocSets may be used to represent something similar to a Digital Bible Library bundle, a Paratext project or a Scripture Burrito. The boundaries of collections are defined globally, at processor instantiation, using selectors.

Selectors

Selectors, together, form a composite, primary key for a collection. A new document that has the same set of selectors as an existing docSet will be added to that docSet. A new document that has a set of selectors that matches no existing docSet will be added to a newly-created docSet. Each selector may be a string or a number, and the range of values may be constrained in various ways:

- for numbers, minimum and/or maximum value
- for strings, regex match
- for numbers and strings, enum membership

The default selectors are

- lang (a string representing the language code of the docSet)

- abbr (a string abbreviation for the docSet)

The id field of a docSet is composed of its selectors. By default the selectors are are separated by underscores, eg *en_ust*.

Subclasses of Proskomma may specify different selectors, and may concatenate those selectors in a different way.

It is also possible to request all the selectors, or one specific selector.

Tags

Tags may be applied to a docSet. It is possible to filter docSets by the presence or absence of tags, and to request that tag information in various ways.

Documents

A docSet always contains at least one document. Fields exist to return a single document, by bookCode, and to return an array of documents, filtered by many criteria.

The DocSet Token Enums

The DocSet stores enums for all the tokens in all the documents in the docSet. Various fields allow introspection of that information.

See the full schema for docSets for more details.

Tags

Tags are strings that can be applied to docSets, documents and sequences. They are intended to support application-defined workflows and categories, eg to denote draft or unchecked documents.

The regular expression that validates tag names is

```
^[a-z][a-z0-9]*(:.+)?$
```

ie a lower-case letter, followed by optional lower-case letters and digits, optionally followed by a colon and an arbitrary string.

No-colon tag names are intended to be used for simple labels. Tag names with colons can be used to implement basic key-value logic (although the underlying implementation is not designed for scale.)

Tags are managed using set logic. This means that it is legal to attempt to add the same tag multiple times, or to attempt to remove a tag that does not exist. Tags are stored as an unordered collection, so applications should make no assumptions about the order in which tags will be returned.

In some cases, tags may be specified when documents are created. Tags may also be added or removed via GraphQL mutations.

Tags can be returned for the type to which they are attached, and may also be used to filter collections.

Documents

A document typically represents the content relating to a single Bible book.

Identification

Each document has a generated id that it unique to the Proskomma instance. The headers of USFM documents (or their equivalents) can be accessed, as key-value tuples or by key. Additional fields exist to parse the id header which typically contains several types of information.

Tags

Tags may be applied to a docSet. It is possible to filter docSets by the presence or absence of tags, and to request that tag information in various ways.

Sequences

Each document contains a main sequence and, optionally, other sequences which are grafted recursively to the main sequence to produce a tree structure. Fields exist to return the main sequence, to return an array of sequences, and to return the number of sequences cheaply.

Fields also exist to return text, table, tree or key-value sequences. (All sequences can be queried as vanilla sequences, but the results may not always be very useful). These fields essentially cast a sequence to a more specific GraphQL type. This casting will fail if, eg, a table is treated as a tree.

Chapter/Verse Lookup

Fields exist to return the content for a chapter, a verse, a range of verses within a chapter and a range that spans chapters, specified in various ways. If a versification file has been added to the docSet, it may be used to query chapter/verse via verse mapping.

Content by chapter/verse

It is also possible to query for all the chapters in the document, or for a single chapter, optionally chunked by verse.

Convenience Fields

Various fields exist to provide easy access to the text of the document, without explicitly destructuring sequences and blocks.

See the full schema for documents for more details.

Sequences

Each document contains one or more sequences. A sequence is a contiguous flow of content. Each document has exactly one main text sequence, to which other sequences may be grafted. For example, titles and footnotes are stored as separate sequences that are grafted to another sequence.

By default, all sequences are treated as text sequences. (This always 'works' although the results may not be very useful in some cases.)

Text Sequences

Text sequences are intended for handling documents containing paragraphs of text. All sequences can be treated as text sequences.

Identification

Each sequence has a generated id that it unique to the Proskomma instance. It also has a type. For the main sequence, which is always a text sequence, that type is 'main'. For non-text sequences the type is 'table', 'tree' or 'kv'. Other types are used for parts of the decomposed text document, such as headings, footnotes and sidebars.

Tags

Tags may be applied to a docSet. It is possible to filter docSets by the presence or absence of tags, and to request that tag information in various ways.

Blocks

In a text sequence, a block corresponds to a paragraph. They are stored in an array, and may be filtered by a large number of criteria.

Item Groups

Sequence content may also be chunked up according to some combination of scopes. This was initially used to split up sequences by chapter and verse, but can be used to destructure any semantic, scope-based markup. This is a relatively expensive operation since the whole sequence is traversed, and because the block-oriented indexes are not optimised for this.

Token Information

Fields exist to return all the unique tokens used in a sequence, and to test whether specific tokens are used.

Convenience Fields

Various fields exist to provide easy access to the text, tokens or items of the sequence, without explicitly destructuring blocks.

See the full schema for text sequences for more details.

Table Sequences

Table sequences are intended for handling tabular data. They repurpose various sequence features originally designed for text sequences to offer reasonable performance via a convenient interface.

Descriptive Fields

Table sequences, like all sequences, have a unique id. It is also possible to query the number of columns, rows and cells. Tags work as for text sequences.

Column Headings

Column headings are stored as tags. A convenience field returns all the column headings in order.

Cells

A cell is a Proskomma block by another name. It contains an array of content items, as well as arrays of rows and columns occupied by the cell. (The storage format supports multiple row and/or column spans although, at present, the input filters do not support spans.) The cells endpoint provides cells as a flat array, as if reading the table from left to right and from top to bottom.

Rows

The rows field chunks up the table cells by rows, and allows filtering by rows, by columns and by content.

See the full schema for table sequences for more details.

Tree Sequences

Tree sequences are intended for handling tree data. (They were initially implemented to handle syntax trees.) They repurpose various sequence features originally designed for text sequences to offer reasonable performance via a convenient interface.

Descriptive Fields

Tree sequences, like all sequences, have a unique id. It is also possible to query the number of nodes. Tags work as for text sequences.

Nodes

A node is a block by another name. Node fields provide easy access to the node id, its parent and children ids, its keys and the items it contains.

Tribos Query Language

Tribos is a DSL for querying tree sequences. Tribos queries look a bit like XPath, and are submitted as a single string. The result is also a string containing serialized JSON. Tribos is in active development, and the most current - if terse - documentation is available via a GraphQL field.

See the full schema for tree sequences for more details.

Key-Value Sequences

Key-value sequences are intended for handling data that is accessed by one or more key, like a card index. They repurpose various sequence features originally designed for text sequences to offer reasonable performance via a convenient interface.

Descriptive Fields

Key-value sequences, like all sequences, have a unique id. It is also possible to query the number of entries. Tags work as for text sequences.

Entries

An entry is a block by another name. Entries can be filtered by primary and secondary keys and by content. The fields of each entry are exposed as item groups.

See the full schema for key-value sequences for more details.

Blocks

In a text sequence, a block corresponds to a paragraph. In other sequences, the same underlying structure is used to represent other structures. In all cases, each block contains several typed arrays that are used to store document data in a succinct format:

- **c**: block **c**ontent (an array of items).
- **bs**: **b**lock **s**cope (one start scope). In text sequences this denotes the type of paragraph.
- **bg**: **b**lock **g**rafts (zero or more grafts). In text sequences this denotes sequences that are attached to the top of the block, such as headings.
- **is**: **i**ncluded **s**copes (zero or more grafts). In text sequences this provides fast lookup of scopes in the paragraph.

- **os**: **o**pen **s**copes (zero or more grafts). In text sequences this caches scopes that have been opened but not closed above, such as the current chapter or verse.
- **nt**: **n**ext **t**oken (an integer). This caches the token count within the sequence.

Low-Level Information

Two fields describe each of the above arrays:

- a field ending in *BL* returns the length in bytes of the array. (This is mainly intended for debugging purposes.)
- a field ending in *L* returns the length in items of the array.

There is also a *dump* field that returns the block content in an arcane but compact string format. (This is mainly intended for debugging purposes.)

Scope Information

The raw scopes in *bs*, *bg*, *is* and *os* may be accessed via a field of that name. The *scopeLabels* field combines these various sources of information to provide an array that is often easier to work with.

Document Content

Content may be accessed as items (ie tokens, scopes and grafts), as tokens (with or without denormalized scope information) and as a single text string. Item and token output may be filtered in many different ways.

See the full schema for blocks for more details.

Item Groups

Internally, sequence content is represented as an array of blocks. This is sometimes a convenient way to access the data, eg when rendering documents by paragraph. For other use cases it may be easier to work with data that has been chunked by some other criterion. Item groups provide a generic type to represent this.

Scope Information

The *scopeLabels* field returns the scopes that are active within each item group (both the scopes used to delimit the item groups and other scopes.)

Document Content

Content may be accessed as items (ie tokens, scopes and grafts), as tokens, and as a single text string.

See the full schema for itemGroups for more details.

Chapter and Verse Indexes

Chapter and chapter-verse indexes provide a way to access content rapidly by book-chapter-verse (BCV).

Direct Access to Indexes

Fields exist to read the start and end location of each BCV unit.

BCV information

Chapter indexes include a field to return the chapter number. For chapter-verse indexes, information is provided about both verses and verse ranges, with mappings in both directions.

See the full schema for chapter indexes and the full schema for chapter-verse indexes for more details.

Items

Items are the basic unit of content in Proskomma. There are three types of items. All items have the same structure:

- a type ('token', 'scope' or 'graft').
- a subType (depending on the type).
- a payload (depending on the type).

Tokens

Tokens contain text:

- the type is 'token'.
- the subType describes the text ('wordlike', 'linespace' or 'punctuation').
- the payload contains the actual text.

Scopes

Scopes enclose other content, and are used to represent a wide range of markup:

- the type is 'scope'.
- the subType is 'start' or 'end'.
- the payload contains a string representation of the scope (strings separated by '/').

Grafts

Grafts connect sequences, and are used to add headings, footnotes and other content:

- the type is 'graft'.
- the subType is the type of graft.
- the payload contains the id of the grafted sequence.

See the full schema for items for more details.

Mutations

Mutations are used to add, remove or modify content. Each mutation takes a number of arguments and returns a boolean success code.

See the full schema for mutations for more details.

1.4 Rendering and Serializing

1.4.1 Serializing to Proskomma Native Format

Proskomma docSets may be serialised as follows:

```
const serialized = pk.serializeSuccinct('eng_kjv');
```

All loaded docSets may be serialized using the *proskomma-freeze* module:

```
const { freeze, thaw } = require('proskomma-freeze');
const frozen = await freeze(pk);

// some time later...

await thaw(pk2, frozen);
```

There are currently no serialization options below the docSet level because, internally, Proskomma generates content enums for each docSet. This means that two serialized documents cannot be merged into one docSet, because the same content in the two documents is likely to be encoded using different enums.

1.4.2 The Proskomma Rendering Model

See the github repo.

1.4.3 Rendering USFM

See the github repo.

1.4.4 Rendering Epubs

See the github repo.

1.4.5 Rendering PDFs

PDF rendering uses PagedJS to produce 'chunked' HTML that can be printed to a PDF file from a browser.

See the github repo.

1.5 Implementation

1.5.1 Proskomma Architecture

1.5.2 Proskomma Succinct Data Representation

1.5.3 The Proskomma Development Cycle

1.6 Getting Involved

1.7 Frequently-Asked Questions

1.7.1 FAQ - What is Proskomma?

How does Proskomma compare to other data formats?

A short answer might be that Proskomma generates the GraphQL response format. However, Proskomma is not a format like USFM or OSIS. Proskomma can import several formats, which it converts to an internal, succinct, binary format. That format can be serialized, but it is not intended to be used outside of Proskomma.

Proskomma is a Scripture Runtime Engine. In a model-view-controller architecture, Proskomma is a model. It manages Scripture-related data, and can provide some or all of that data to the view. That data is always formatted as a GraphQL response, but the 'shape' of that response can vary a lot, depending, on the query, and according to the needs of the application.

The most fundamental example of this is that canonical text may be chunked up by paragraph, or by verse, or in other, arbitrary ways. Most Scripture-oriented formats choose either paragraphs or chapter/verse. Both options are extremely convenient for some use cases, and somewhat awkward for other use cases. An application using Proskomma can choose a different content structure in each query, regardless of the input format.

Is Proskomma a database?

Yes, in the high-level sense that Proskomma provides a view onto data via a query language.

No, in that it is not designed to handle high levels of concurrency or terabytes of data. Out of the box, Proskomma does not even persist data between sessions (although there are utilities to implement this).

In use, Proskomma is more like a document-based DBMS such as MongoDB than a table-based DBMS such as MySQL. However, it provides multiple views of document structure via the GraphQL interface.

Can I use Proskomma without a server?

Yes - this is currently the most common way to use Proskomma. The GraphQL interface may be accessed by calling an async method on the Proskomma instance.

Can I use Proskomma via a server?

Yes - the Proskomma Node Express project provides proof of concept of this.

Can I use Proskomma with <My Favorite UI Framework>

Almost certainly. Most UIs to date have used React, but Proskomma itself is agnostic regarding UI design. The only requirements are support for modern Javascript and a plan to manage async queries to Proskomma.

Does Proskomma handle verse mapping?

Yes, via versification files like those used in Paratext.

Does Proskomma handle alignment?

Not specifically, because the Bible-tech world has not agreed on one way to do alignment. However, Proskomma provides a set of generic tools that have been used to implement alignment.

1.8 The Proskomma Ecosystem

1.8.1 Diegesis Ionic App

See the github repo.

1.8.2 Chaliki Electron App

See the github repo.

1.8.3 Proskomma Node Express Server

See the github repo.

1.8.4 Koniortos Mobile App

See the github repo.

1.9 "Powered by Proskomma"