Contents

1 Design notes

Let's start off with how BIEL sends a document generation request that specifies what (arbitrary) combination of resources a user would like included in the final PDF document.

1.1 JSON document generation request design for web client (BIEL)

Requirement: Allow creating a document out of any combination of resources from any supported (in translations.json) language.

At present we are at the stage of matching the functionality of the prior system with respect to granularity of selection of bible material, i.e., book-level granularity.

Currently the JSON looks like this, for example:

```
{ "resources":
  { "lang_code": "am", "resource_type": "ulb", "resource_code": "gen" },
  { "lang_code": "lpx", "resource_type": "tn", "resource_code": "exo" },
  ...
}
```

Each element in the JSON dictionary represents a resource. The whole dictionary represents all the resources you want in the final typeset document in the order you want them. Note that last point, the order. That could of course be changed, but I am for now making an assumption that BIEL's wizard would compose a JSON document generation request having resources in the order the user requested the documents resources to be represented.

1.1.1 TODO Question: What is the smallest level of resource request granularity we want: book, chapter, or verse?

In the resource entries below note that I've changed `resource_code` to `book_code` just because it might be a better name than `resource_code`.

Perhaps book level granularity is sufficient, but just covering it here in case. More granular than book level would require a design that I can envision but would require interrogating the resource itself rather than just the translations.json API.

So, the following rest of this headline is probably a diversion, but here it is just in case you want finer than book granularity.

If chapter is the finest granularity of a resource request, the JSON could be:

```
{ "resources":
  { "lang_code": "am",
    "resource_type": "ulb",
    "book_code": "gen",
    "book_chapter": "1" }, // Get just chapter 1
  { "lang_code": "lpx",
    "resource_type": "tn",
    "book_code": "exo",
    "book_chapter": "" }, // Get the whole book
  ...
}
```

If verse is the finest granularity of a resource request, the JSON could be:

```
{ "resources":
  { "lang_code": "am",
    "resource_type": "ulb",
    "book_code": "gen",
    "book_chapter": "1",
    "verse_start": "1",
    "verse_end": "3" }, // Get chapter 1, verse 1-3
  { "lang_code": "am",
    "resource_type": "ulb",
    "book_code": "gen",
    "book_chapter": "1",
    "verse_ranget": "1-1" }, // Or, Get chapter 1, verse 1
  { "lang_code": "am",
    "resource_type": "ulb",
    "book_code": "gen",
    "book_chapter": "1",
    "verse_ranget": "1-3" }, // Or, Get chapter 1, verse 1-3
  { "lang_code": "am",
    "resource_type": "ulb",
    "book_code": "gen",
    "book_chapter": "1",
    "verse_ranget": "1-3,5" }, // Or, Get chapter 1, verse 1-3 and verse 5
  { "lang_code": "lpx",
    "resource_type": "tn",
    "book_code": "exo",
    "book_chapter": "2" }, // Get chapter 2
  { "lang_code": "lpx",
    "resource_type": "tn",
    "book_code": "exo",
    "book_chapter": "" }, // Get the whole book
  ...
}
```

As said before in a slightly different context, ideally from a user experience perspective, BIEL would need to know what chapters or verses are available so as not to disappoint the user. Nevertheless, the system is being designed to gracefully handle such disappointments as I know this is a requirement.

1.1.2 **TODO** for Craig: a license for the `Interleaved_Resources_Generator` project

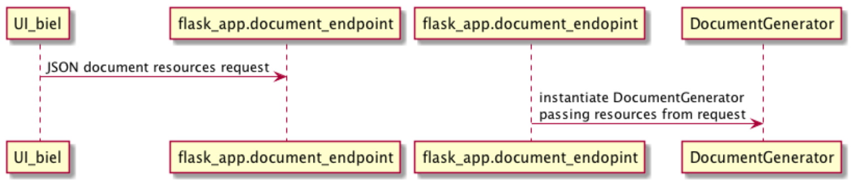
I need a license emailed to me that I can check in to the repo. Or you can send me a link to the license and I'll get it there. Thanks!

1.1.3 Example request from `test_flask.py`

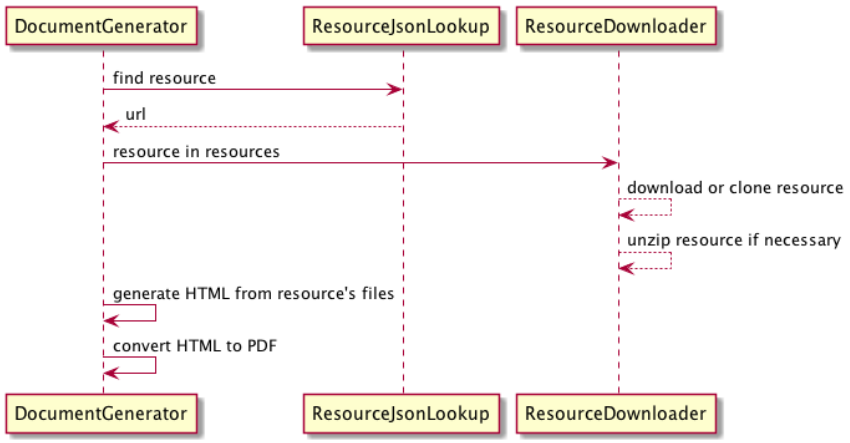
Example python client request using resources JSON dictionary to submit an API call to generate a document composed of resources.

```
import json
import requests

payload = {}
payload["resources"] = [
    {"lang_code": "am",
     "resource_type": "ulb",
     "resource_code": ""},
    {"lang_code": "erk-x-erakor",
     "resource_type": "reg",
     "resource_code": "eph"},
    {"lang_code": "ml",
     "resource_type": "ulb",
     "resource_code": "tit"},
    {"lang_code": "ml",
     "resource_type": "obs-tq",
     "resource_code": ""},
    {"lang_code": "mr",
     "resource_type": "udb",
     "resource_code": "mrk"},
]


res = requests.post("http://localhost:5005/api/v1/document", json=json.dumps(payload)
if res.ok:
    print(res.json())
```

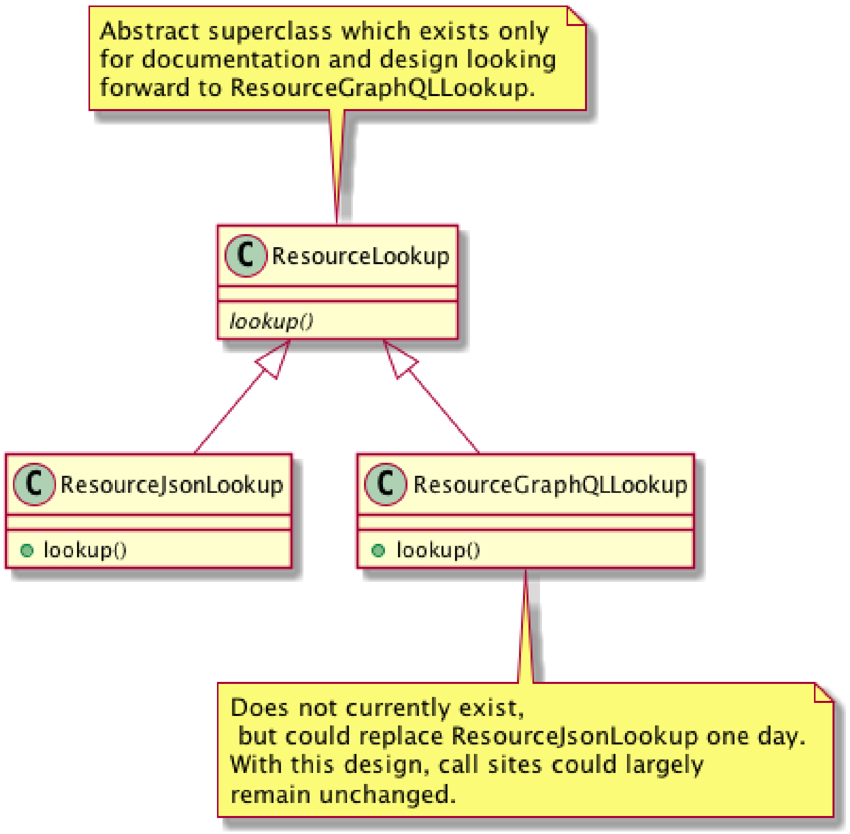
1.2 Interactions at a high level

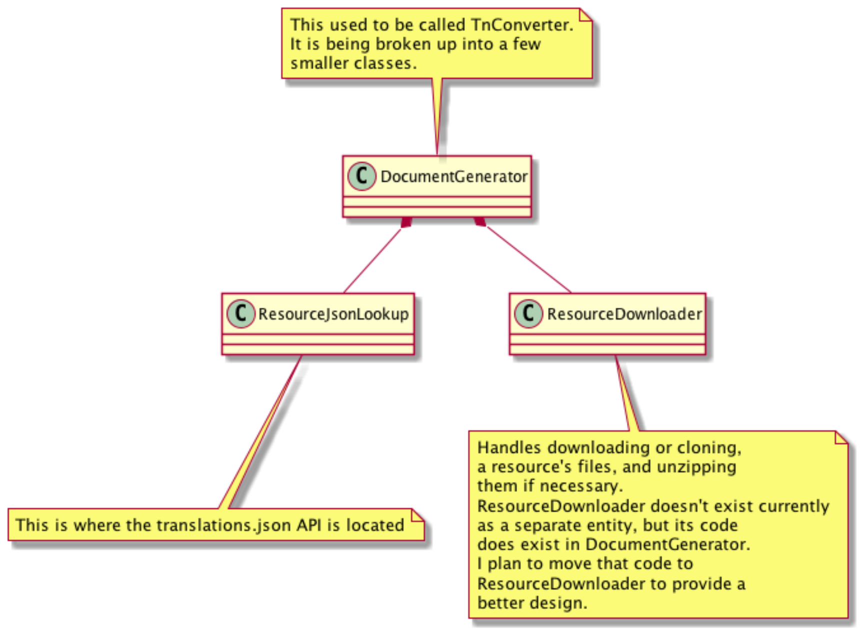

UI_biel
flask_app.document_endpoint
flask_app.document_endopint
DocumentGenerator
JSON document resources request
instantiate DocumentGenerator
passing resources from request

DocumentGenerator passes back a JSON dict containing any messaging and the eventual location of the generated document for display to the requesting user (by BIEL).



Note that, of course, DocumentGenerator iterates through the resources and submits each resource to be found and provisioned, so "find resource" in the image above is happening in a loop through each resource. Similary for ResourceDownloader.


Abstract superclass which exists only for documentation and design looking forward to ResourceGraphQLLookup.
ResourceLookup
lookup()
ResourceJsonLookup
lookup()
ResourceGraphQLLookup
lookup()
Does not currently exist, but could replace ResourceJsonLookup one day. With this design, call sites could largely remain unchanged.


This used to be called TnConverter.
It is being broken up into a few
smaller classes.
DocumentGenerator
ResourceJsonLookup
ResourceDownloader
This is where the translations.json API is located
Handles downloading or cloning,
a resource's files, and unzipping
them if necessary.
ResourceDownloader doesn't exist currently
as a separate entity, but its code
does exist in DocumentGenerator.
I plan to move that code to
ResourceDownloader to provide a
better design.

1.3 What works currently

1. Making a request for document generation to the web service (flask) running.
2. The resources that comprise the document generation request can handle a book-level of request granularity at present.
3. Resources are found and provisioned to disk (but not yet typeset into a final document).

Files involved: `flask_app.py`, `resource_lookup.py`, `document_generator.py` (and `config.py`, `file_utils.py`, `url_utils.py`).

1.4 Concept of resource in system

What follows is more than you really need to know, but I feel it is worth an explanation since you are observing the evolution of a code base that is very much in flux and far from complete.

At present, because the final form of the data structures are not yet solidified I have found it advantageous to pass around a dictionary that is a reification of the original JSON document generation request (this is seen in both `ResourceJsonLookup` and `DocumentGenerator`. This has been nice because prior to refactoring the legacy portion of this system, `export_md_to_pdf.py` (which is now named `document_generator.py`), I can update said dictionary with fields as I need them. A dictionary during this stage of development provides the flexibility to not commit to a design yet. Adding additional key/value pairs to the dictionary like `resource_dir`, `resource_file_format`, etc. is simple. These additional fields store data that are important for a resource to know about itself so that its related files can be provisioned to disk and subsequently found and used by the document generation processes themselves (for things like Markdown to HTML conversion, HTML to PDF conversion, etc..).

The use of the dictionary may (likely) change as the proper data structures emerge from the evolving design. Of course, this is solution domain stuff and so not something you should be concerning yourself with as it doesn't have to do with requirements. Things will continue to change from iteration to iteration.

1.5 Docker container

There isn't much to say about the docker container except that it provides the runtime environment, obviously. The only significant new detail is that flask can be specified to run on a particular IP and port (seen in `docker-compose.yaml`) which BIEL will know and use when submitting requests.

In a later iteration toward the end, flask will presumably be load balanced. Further, to protect its pool of workers from being tied up by long running client requests from BIEL, one can adopt an architecture such as the one described in the next paragraph.

nginx in front of gnunicorn in front of flask could be put in place to handle load balancing incoming front end requests from BIEL. To learn why you might do something like that please see this stackoverflow answer

I am not bothering myself with this at all right now, just mentioning it. There are plenty of other architectures that could be used when we get there.

1.6 (Bonus/optional material) Convenience web service endpoints for BIEL UI to call (if desired)

A nice property for a system like this to have is a ground truth data source so that front end (BIEL) and back end are on same page about what resources are available.

For now, that ground truth data source is the latest copy of translations.json that `ResourceJsonLookup` obtains and keeps fresh to within each 24 hour window. (This works fine, but I may make this more sophisticated later).

Toward that goal of one ground truth data source for BIEL and this system to coincide with this system provides to BIEL a couple of web app endpoints that it can request data from to populate its dropdown menu's in BIEL's document request wizard. These endpoints were easy to make, so I am providing them.

It would be a bad user experience for BIEL users to be able to request a resource which does not exist (but that is outside my scope per requirements). For now, if these endpoints are used, we at least make sure only languages that are provided in (the same version of) translations.json are available for selection. Maybe later we'll go further and actually provide endpoints that return the resources available per language also. If those endpoints were built, they could also be used by BIEL to populate resource type and book dropdown menus. Just putting it out there though we haven't talked about it.

The system is being designed to gracefully handle non-existent requested resources per requirements, but naturally you'd want to avoid this if possible.

These endpoints were quick to create and were used in part to test flask and jsonpath performance (you'll note that it is one place I don't use jsonpath since performance in this one case was unacceptable). So consider these endpoints a happy byproduct of development, but that could be expanded to provide a better overall user expeience if desired.

1.6.1 Example client call to get all language codes by themselves

Example client call from `test_flask.py`:

```
import json
import requests

res = requests.get("http://localhost:5005/api/v1/language_codes")
if res.ok:
print(res.json()) # Presumably, BIEL'll display it in a drop down menu or similar.
```

1.6.2 Get all language code, language name pairs

Example client call from `test_flask.py`.

```
import json
import requests

res = requests.get("http://localhost:5005/api/v1/language_codes_and_names")
if res.ok:
    print(res.json()) # Presumably, BIEL'll display it in a drop down menu or similar
```