tx-manager Documentation

Release

unfoldingWord

Apr 27, 2018

Contents

master: develop:

CHAPTER 1

tX Overview

tX (translationConverter) is a conversion tool for the content in the Door43 Conent Service (DCS). The goal is to support several different input formats, output formats, and resource types.

See the documentation at https://tx-manager.readthedocs.io/en/latest.

Issue queue maintained at https://github.com/unfoldingWord-dev/door43.org/issues.

Setting Up a Developer Environment (CLI Version)

Satisfy basic depedencies:

```
git clone git@github.com:unfoldingWord-dev/tx-manager.git
sudo apt-get install libssl1.0.0 python-pip
pip install virtualenv
```

We recommend you create a Python virtual environment to help manage Python package dependencies:

```
cd tx-manager
virtualenv venv
```

Now load that virtual environment and install dependencies:

```
source venv/bin/activate
pip install -r requirements.txt
```

Set AWS variables:

```
export AWS_DEFAULT_REGION=us-west-2
export AWS_REGION=us-west-2
```

Run the test suite:

```
python test-setup.py test
```

Or, to run a single test, run:

```
python -m unittest tests.client_tests.test_client_webhook
```

Optionally to do Integration tests on 'test' site, first deploy tx-manager to test:

```
apex deploy -p test -e test
```

Now you can run the integration test(s) (see run_integration_tests for setup steps and help):

```
./scripts/run_integration_tests.sh test_ts_mat_conversion
```

tX Pipeline

1. Gogs
2. Webhook
3. Request Job
4. Start Job
5. [CONVERTER]
6. Callback
7. Door43 Deploy

Definitions

The following placeholders are used in examples in this document:

- <repo> - the machine name of a Gog's repo. This is used in the URL for the repo, such as en-obs
- <user> - the user or organization that the Gog's repo belongs to, such as richmahn (user) or door43 (org)
- <commit> - The 10 character hash string that represents the commit (revision) that is being processed

How it Works

Request Conversion Job

Using the Pipeline and the corresponding numbers above, this describes each part of the pipeline and how each are integrated, both with each other as well as the AWS Services that are used.

NOTE: This gives URLs and bucket names for test. For development, replace the *test-* prefix from domain or bucket name with *dev-*. For production, remove the *test-* prefix from domain or bucket name.

1. Gogs (Git website)

When a repository is updated on Gogs, the commit triggers all webhooks in the repo's settings. One of those webhooks, which our copy of Gogs sets up automatically for every new repo, is a call to https://test-api.door43.org/client/webhook (API Gateway -> Lambda function).

2. Webhook (Lambda function - API Gateway triggered)

   The webhook triggered in Gogs (#1) sends the commit payload to the AWS API Gateway *client* stage and the *webhook* method which triggers the client_webhook Lambda function.

   The webhook function expects the following variables in the payload:

   - data - the commit payload from Gogs
   - api_url* - the base URL to the tX Manager API (e.g. https://test-api.door43.org)
   - pre_convert_bucket* - the S3 bucket in which to put the zip file of the files to be converted (e.g. tx-webhook)
   - cdn_bucket* - the S3 bucket in which the zip file of the converted files is to be found in client_callback (e.g. cdn.door43.org)
   - gogs_url* - the URL to the Gogs site to verify user token (e.g. https://git.door43.org)
   - gogs_user_token* - a user token of a valid user to prove they are a user so we can track job requests (for the webclient we just have one user token for all requests, given by the API Gateway)

   these variables are set up in the 'client' Stage Variables <https://us-west-2.console.aws.amazon.com/apigateway/home?region=us-west-2#/apis/94c6v76xoh/stages/client>, so dev and prod gateways can have different variables

   The client_webhook function is responsible for standardizing both a manifest.json file and the resource containers from all types of repos committed to Gogs, and it will call a preprocessor (e.g. TsObsMarkdownPreprocessor) to handle this. Converters (#4) expect the files to be converted to be in a flat-level zip file, where all files to be converted (with the input file extension) are one file per chapter (Bible, OBS) and in alphabetical order for logging and display purposes. Once the files are zipped up and the zip file put at https://test-cdn.door43.org/temp/<repo>/<commit>, the client webhook function requests a job by posting a request to https://test-api.door43.org/tx/job and exits.

3. Request Job (Lamdbda function - API Gateway triggered)

   Request Job is triggered through a call to the AWS API Gateway, running the request_job lambda function. This function expects the following variables in the payload:

   - gogs_url* - the URL to the Gogs site to verify user token (e.g. https://git.door43.org)
   - api_url* - the base URL to the tX Manager API (e.g. https://test-api.door43.org)
   - data - information about the job to performed. It contains the following variables:
     - gogs_user_token - a user token of a valid Gogs user
     - cdn_bucket - the S3 bucket in which the zip file of the converted files is to placed
     - source - The URL of the archive of files to convert (e.g. https://s3-us-west-2.amazonaws.com/test-tx-webhook/preconvert/0038b1d1-bf3b-11e6-8481-ed2b5603783b.zip)
     - resource_type - The resource type (e.g. obs, ulb, udb, etc.)
     - input_format - The input format of the files (e.g. md)
     - output_format - The desired output format (e.g. html)
   - these variables are set up in the* 'tx' Stage Variables *, so dev and prod gateways can have different variables.*

From the above information, tX Manager's request_job function will determine what converter to use for this job and will save this job request to the *jobs* table. It will then invoke the tX Manager Start Job lambda function.

4. Start Job (a) (Lamdbda function - DynamoDB tx-job table insert triggered)

   The Start Job lambda function is triggered by a job being inserted into the DynamoDB tx-job table (Thus is NOT triggered through a call through the API. This is to separate the Request Job from the Start Job due to the 5 minute limit of execution time of a Lambda function)

   This function will load the given record from the DB and populate a TxJob object. It will then send this to the converter determined in #3 from its input and output formats. A call to the converter is then made.

5. [CONVERTER] (Lamdbda function - tX Manager triggered)

   Each converter is responsible for converting a given input file type to a given output file type. It also can have one or more resource types it converts. It expects the URL of a zip file which it then downloads and unzips. It then converts all the files to another zip file, converting the files of the given input type to the given output type, and copies all other files as they are to the new archive. It uploads the archive to the given S3 bucket and file path.

   It also can perform checks at this point if there any warnings or errors and return those in the JSON object returned to the Start Job function (#4)

4. Start Job (b) (Lamdbda function - Return from [CONVERTER] #5)

   Once the CONVERTER returns a status of warnings and errors (if any), the Start Job function calls the call back URL if one was given so the client can know the job was completed and if it was successful or not.

6. Callback (Lamdbda function - API Gateway triggered)

   When the callback function is called, the client looks to see if the job was a success and if it was, unzips the new archive and puts its contents in the test-cdn.door43.org bucket with the key prefix of *u/<user>/<repo>/<commit>*. It puts the status of the build into a file and uploads to the same bucket with the key *u/<user>/<repo>/<commit>/build_log.json*.

7. Deploy to Door43 (Lamdbda function - S3 modified file triggered)

   The uploading of build_log.json in #6 triggers the Door43 Deploy function.

   The Door43 Deploy function is what moves the HTML files converted by #5 and placed in the CDN bucket in #6 to door43.org and templates it based on the door43.org layouts. It also generates header, status and navigation portions of the pages for each revision.

Register Conversion Module

In order for tX Manager to know about a conversion module and to assign a conversion request to the module, it must be registered. To register a module, it must make a call to the API Gateway with the URL https://test-api.door43.org/tx/register. It expects the following variables:

- name - the Lambda function name of the converter, usually in the form of tx-<input>2<output>_convert
- type - the type of the module, usually "conversion"
- input_format - the input format accepted by the conversion, which is the extension of the file, such as "md"
- output_format - the output format of the files to be generated, which is the extension of the file, such as "html"
- resource_types - the resource type(s) accepted by the converter, such as "obs"

See tx-md2html_register Lambda function. for an example of a module registering itself.

Setting up as deployed in virtual environment

In IntelliJ terminal, switch to virtual environment and install requirements.

```
source ~/venv/txml/bin/activate
./install-requirements.sh
```

Deploying your branch of tx-manager to AWS

For developing the tx-manager library which this repo uses for every function, you can deploy your code to a test AWS environment with apex by doing the following:

- Copy project.test.json.sample to project.test.json
- Edit project.test.json and change <username> and <branch> to your tx-manager branch
- Install apex from http://apex.run/#installation
- Set up your AWS credentials as specified at http://apex.run/#aws-credentials
- Run *apex deploy –env test* to deploy all functions, or *apex deploy –env test [function-name]* for a single function

For more information on using –env to specify a project json file, see https://github.com/apex/apex/blob/master/docs/projects.md#multiple-environments

CHAPTER 2

tX Architecture

This document explains the layout of the translationConvertor (tX) conversion platform and how the components of the system should interact with one another.

If you just want to *use* the tX API, see tX API Example Usage

Keep reading if you want to contribute to tX.

Goals

tX is intended to be a conversion tool for the content in the Door43 Platform. The goal is to support several different input formats, output formats, and resource types.

Development goals are:

- Keep the system modular, in order to:
- Encourage others to contribute and make it simple to do so
- Contain development, testing, and deployment to each individual component
- Constrain feature, bugfixes, and security issues to a smaller codebase for each component
- Continuous Deployment, which means
- Automated testing is required
- Continuous integration is required
- Checks and balances on our *process*
- RESTful API utilizing JSON

Infrastructure

Overview

All code for tX is run by AWS Lambda. The AWS API Gateway service is what provides routing from URL requests to Lambda functions. Data and any required persistent metadata are stored in AWS S3 buckets. This is a "serverless" API.

Developers use Apex, Travis CI, and Coveralls.

Permissions (mostly for accessing S3 buckets) are managed by the `role` assigned to each Lambda function.

Modules may be written in any language supported by AWS Lambda (including some that are available via "shimming"). As of July, 2016, this list includes:

- Java (v8)
- Python (v2.7)
- Node.js (v0.10 or v4.3)
- Go lang (any version)

Modules MUST all present an API endpoint that the other components of the system can use. Modules MAY present API endpoints that the public can use.

Background: Issue for tX creation at https://github.com/unfoldingWord-dev/door43.org/issues/53.

Separating Production from Development and Test

We want our code to not know/care if it is running in production, development or test environments. Yet there are plenty of variables and locations that data and files are stored that vary from the three, such as different bucket names between our two AWS accounts, since all bucket names on AWS must be unique.

So that the clients, tx-manager and convert modules don't have to worry about this, everything that varies from environment will be set up in the API Gateway Stage Variables. These are variables we set up in AWS for a particular API's URL. Along with the payload sent by the requesting client, these variables will also be put into the "event" variable in the Lambda handle function.

For example, such variables may be (test | development | production):

- cdn_bucket = "test-cdn.door43.org" | "dev-cdn.door43.org" | "cdn.door43.org"
- api_bucket = "test-api.door43.org" | "dev-api.door43.org" | "api.door43.org"
- door43_bucket = "test-door43.org" | "dev-door43.org" | "door43.org"
- api_url = "https://test-api.door43.org" | "https://dev-api.door43.org" | "https://api.door43.org"
- gogs_user_token = "<a user token from test.door43.org:3000>" | "<a user token from dev.door43.org:3000>" | "<a user token from git.door43.org>"
- gogs_username = "<username of the above user_token>"
- env = "test" or "dev" or "prod" (just in case you want to still do something different based on environment in your code)

Test Environment

The test environment should use the WA AWS account. There are 3 test buckets that have been created that mirror the production buckets:

- test-api.door43.org - for tx-manager to manage data for tX (only /tx namespace should be used) (public access disabled on this)
- test-cdn.door43.org - for conversion modules to upload their output to (only /tx namespace should be used) (public access enabled on this)
- test-door43.org - For Jekyll and /u generated files to upload to (public access enabled on this)

Use *apex deploy* to upload code for lambda functions to test environment.

Development Environment

The development environment should use the Door43 AWS account. There are 3 development buckets that have been created that mirror the production buckets:

- dev-api.door43.org - for tx-manager to manage data for tX (only /tx namespace should be used) (public access disabled on this)
- dev-cdn.door43.org - for conversion modules to upload their output to (only /tx namespace should be used) (public access enabled on this)
- dev-door43.org - For Jekyll and /u generated files to upload to (public access enabled on this)

The `develop` branch for each repo should automatically deploy to this account and make use of the above buckets.

Production Environment

The production environment should use the Door43 AWS account. The production buckets are:

- api.door43.org - for tx-manager to manage data for tX (only /tx namespace should be used) (public access disabled on this)
- cdn.door43.org - for conversion modules to upload their output to (only /tx namespace should be used) (public access enabled on this)
- door43.org - For Jekyll and /u generated files to upload to (public access enabled on this)

The `master` branch for each repo should automatically deploy to this account and make use of the above buckets.

Modules

Every part of tX is broken into components referred to as `tX modules`. Each tX module has one or more functions that it provides to the overall system. The list of tX modules is given here, with a full description in its respective heading below.

- *tX Webhook Client* - Handles webhooks from git.door43.org (Gogs) to format the repo files, massaging them based on resource and format into a flat directory structure and zips it up to invoke a job request with the *tX Manager Module*.
- *tX Manager Module* - Manages the registration of *conversion modules* and handles job requests for conversions. Makes a callback to the *client* when conversion job is complete.
- *tX Authorization Module* (actually just the python-gogs-client)

- *tX Conversion Modules* - modules that handle the conversion from one file format to another of one or more resources
- *tX Door43 Module* - When a conversion job is completed, it is invoked to make the converted file accessible through the door43.org site, setting up a new revision page for the corresponding Gogs repository. It also maintain stats on the particular project or project revision, such as views and stars

tX Manager Module

The tX Manager Module provides access to three functions:

- Maintains the registry for all *tX Conversion Modules*
- Authorization for requests via the `tx-auth module <#tx-authorization-module>‘__
- Accepts user credentials via `HTTP Basic Auth` (over HTTPS) to verify the calling client is a gogs user
- Counts requests made by each token [not implemented]
- Blocks access if requests per minute reaches a certain threshold [not implemented]
- Handles the public API paths that a tX Convertion modules register
- Job queue management. Accepts job requests with parameters given to it, the most important being a URL to a zip file of the source files, the resource type, input format, and output format. These files must be in a flat ZIP file (no sub-directories, at least not for the files of the input format), conforming to what the tX Converter expects
- Makes a callback to client when job is completed or has failed, if a callback URL was given by the client when the job was requested

The tX manager does NOT concern itself with nor has knowledge of: * `git.door43.org` repositories * `door43.org` pages

tX Authorization Module

The tX Authorization Module is an `authorization` module for the tX system. In reality, this is just the python-gogs-client. The `tx-manager` module uses it to perform authorization of request. The module handles the following:

- Grants access to the API based on a Gogs user token

tX Conversion Modules

Conversion modules include (some are still to be implemented):

- tx-md2html - Converts Markdown to HTML (obs, ta, tn, tw, tq)
- tx-md2pdf - Converts Markdown to PDF (obs, ta, tn, tw, tq)
- tx-md2docx - Converts Markdown to DOCX (obs, ta, tn, tw, tq)
- tx-md2epub - Converts Markdown to ePub (obs, ta, tn, tw, tq)
- tx-usfm2html - Converts USFM to HTML (bible)
- tx-usfm2pdf - Converts USFM to PDF (bible)
- tx-usfm2docx - Converts USFM to DOCX (bible)
- tx-usfm2epub - Converts USFM to ePub (bible)

Each conversion module accepts a specific type of text format as its input and the module returns a specific type of output document. For example, there is a md2pdf module that converts Markdown text into a rendered PDF. The conversion modules also require that you specify the resource type (e.g. obs, ta, tn, tw or tq), which affects the formatting of the output document.

Input Format Types

There are currently two accepted input format types:

- Markdown -`md`
- Unified Standard Format Markers - `usfm`

A few notes on input formatting:

- Conversion modules *do not do pre-processing of the text*. The data supplied must be well formed.
- Conversion modules expect *a single file* either:
- A plaintext file of the appropriate format (`md` or `usfm`).
- A zip file with multiple plaintext files of the appropriate format.

In the case of a zip file, the conversion module should process the files in *alphabetical* order. According to our `obs` file naming convention and the `usfm` standard, this process should yield the correct output in both cases.

Output Format Types

For each type of input format, the following output formats are supported:

- PDF - `pdf`
- DOCX - `docx`
- HTML - `html`

Resource Types

Each of these resource types affects the expected input and the rendered output of the text. The recognized resource types are:

- Open Bible Stories - `obs`
- Scripture/Bible - `bible`
- translationNotes - `tn`
- translationWords - `tw`
- translationQuestions - `tq`
- translationAcademy - `ta`

Available Conversion Options

Conversion modules specify a list of `options` that they accept to help format the output document. Every conversion module MUST support these options:

- `"language": "en"` - Defaults to `en` if not provided, MUST be a valid IETF code, may affect font used

- `"css": "http://some.url/your_custom_css"` - A CSS file that you provide. You can override or extend any of the CSS in the templates with your own values.

Conversion modules MAY support these options:

- `"columns": [1, 2, 3, 4]` - Not available for `obs` input
- `"page_size": ["A4", "A5", "Letter", "Statement"]` - Not available for HTML output
- `"line_spacing": "100%"`
- `"toc_levels": [1, 2, 3, 4, ...]` - To specify how many heading levels you want to appear in your TOC.
- `"page_margins": { "top": ".5in","right": ".5in","bottom": ".5in", "left": ".5in" }` - If you want to override the default page margins for PDF or DOCX output.

Deploying Modules

Each module is initially deployed to AWS Lambda via the `apex` command. After this, Travis CI is configured to manage continuous deployment of the module (see Deploying to AWS from Travis CI).

Continuous deployment of the module should be setup such that:

- the `master` branch is deployed to `production` whenever it is updated
- the `develop` branch is deployed to `development` whenever it is updated

The deployment process looks like this:

- Code in progress lives in a feature-named branch until the developer is happy and automated tests pass.
- Code is peer-reviewed, then
- Merged into `develop` until automated testing passes and it integrates correctly in `development`.
- Merged into `master` which triggers the auto-deployment

Registering a Module

Every module (except `tx-manager`) MUST register itself with `tx-manager`. A module MUST provide the following information to `tx-manager`:

- Public endpoints (for `tx-manager` to present)
- Private endpoints (will not be published by `tx-manager`)
- Module type (one of `conversion`, `authorization`, `utility`)

A conversion module MUST also provide:

- Input format types accepted
- Output format types accepted
- Resource types accepted
- Conversion options accepted

Example registration for `md2pdf`:

Request

```
POST https://api.door43.org/tx/module

{
    "name": "tx-md2pdf_convert",
    "version": "1",
    "type": "conversion",
    "resource_types": [ "obs", "bible" ],
    "input_format": [ "md" ],
    "output_format": [ "pdf" ],
    "options": [ "language", "css", "line_spacing" ],
    "private_links": [ ],
    "public_links": [
        {
            "href": "/md2pdf",
            "rel": "list",
            "method": "GET"
        },
        {
            "href": "/md2pdf",
            "rel": "create",
            "method": "POST"
        },
    ]
}
```

Response:

```
201 Created

{
    "name": "md2pdf",
    "version": "1",
    "type": "conversion",
    "resource_types": [ "obs", "bible" ],
    "input_format": [ "md" ],
    "output_format": [ "pdf" ],
    "options": [ "language", "css", "line_spacing" ],
    "private_links": [ ],
    "public_links": [
        {
            "href": "/md2pdf",
            "rel": "list",
            "method": "GET"
        },
        {
            "href": "/md2pdf",
            "rel": "create",
            "method": "POST"
        },
    ]
}
```

tX Webhook Client

The tX Webhook Client is a client to tX. The purpose of this client is to pre-process the `git` repos from Gogs' webhook notifications, send them through tX, and upload the resulting HTML files to the `cdn.door43.org` bucket.

The process looks like this:

When a Gogs webhook is triggered: * Accepts the default webhook notification from `git.door43.org` * Gets the data from the repository for the given commit (via HTTPS request that returns a zip file) * Identifies the Resource Type (via name of repo or `manifest.json` file) * Formats the request (turns the repo into valid Markdown or USFM file(s), then creates a zip file with the files being in the root of the archive) * Sends the valid data (in zip format) through an API call to the *tX Manager Module*, requesting HTML output, which it then should get a confirmation (JSON) that the job has been queued ('status' = 'requested') * Uploads an initial `build_log.json` file to the `cdn.door43.org` bucket as `u/<owner>/<repo>/<commit>/build_log.json` with information returned from the call to the tX Manager (this file will be updated when job is completed) * Uploads the repo's `manifest.json` file to the `cdn.door43.org` bucket as `u/<owner>/<repo>/<commit>/manifest.json` * Returns its own JSON response which will be seen in the Gogs' webhook request results, stating the request was made, the source ZIP and the expected output ZIP locations

When callback is made: * Extract each file from the resulting output ZIP file to the `cdn.door43.org` bucket with the prefix key of `u/<owner>/<repo>/<commit>/` * Updates the `u/<owner>/<repo>/<commit>/build_log.json` in the `cdn.door43.org` bucket with the information given by tX Manager through the callback (e.g. conversion status, log, warnings, errors, timestamps, etc.)

tX Webhook Client does NOT concern itself with: * Converting files for presentation on `door43.org`

tX Door43 Module

The tX Door43 Module contains processes that will update the `door43.org` bucket/site when conversion jobs are completed. It works behind the scenes, so is not an API. Its tasks include:

- convert the files for presentation on `door43.org` when a conversion job is completed and files have been deployed to the `cdn.door43.org` bucket, applying a template and other styling and JavaScript, and deploy them to the `door43.org` bucket, prefixed with `u/<owner>/<repo>/<commit>`
- Update stats of a project or revision such as views, followers and stars from `git.door43.org`

Including Python Packages in a Lambda Function

Requirements for a Python script need to reside within the function's directory that calls them. A requirement for the `convert` function should exist within `functions/convert/`.

The list of requirements for a function should be in a `requirements.txt` file within that function's directory, for example: `functions/convert/requirements.txt`.

Requirements *must* be installed before deploying to Lambda. For example:

```
pip install -r functions/convert/requirements.txt -t functions/convert/
```

The `-t` option tells pip to install the files into the specified target directory. This ensures that the Lambda environment has direct access to the dependency.

If you have any Python files in subdirectories that also have dependencies, you can import the ones available in the main function by using `sys.path.append('/var/task/')`.

Lastly, if you install dependencies for a function you need to include the following in an `.apexignore` file:

```
*.dist-info
```

For Reference

There is a similar API that has good documentation at https://developers.zamzar.com/docs. This can be consulted if we run into blockers or need examples of how to implement tX.

CHAPTER 3

License

CHAPTER 4

Indices and tables

- genindex
- modindex