Resource Container Documentation

Release 0.2

Door43

Apr 11, 2017

Contents

This site is the official documentation for Door43's Resource Container specification. Resource Containers (RCs) are the building blocks of content in our Door43 eco-system.

Contents:

CHAPTER 1

Resource Container Structure

Resource containers (RCs) exist as directories. They may be optionally compressed or packaged so long as the compressed file follows standard naming conventions for the file extension. For example:

- a zipped RC would end in `.zip`,
- a tarred RC would end in `.tar`,
- a tarred+bzip2 RC would end in `.tar.bz2`

A git repository is also a valid way to store RCs.

Note: When naming an RC directory or repository the best practice is to use a combination of the resource and project identifiers e.g. `en-ulb-gen`. If the RC contains more than one project just use the resource identifier with an optional *Slug* formatted qualifier e.g. `en-ulb-nt` where `nt` is the custom qualifier denoting the New Testament.

Directory Structure

RCs have a folder structure like the following:

```
my_resource_container/
    |-.git/
    |-.apps/
    |-LICENSE.md
    |-manifest.yaml
    |-content/
```

- `.git`: only exists when the RC is stored in a git repository.
- `.apps`: is where applications can store custom meta data about the RC. See *App Meta* for more information.
- `LICENSE.md`: contains the appropriate license information for the RC.
- `manifest.yaml`: is the RC *Manifest File*.

- `content`: contains the project files. The name of this directory is subject to the *Manifest File*. It is also possible for there to be multiple directories or excluded altogether.

Project Directory

The folder structure of the project directory in RCs is mostly the same across RC types. The most common structure is indicated below:

```
content/
    |-config.yaml
    |-toc.yaml
    |-front/
    |-01/
    |    |-title.txt
    |    |-sub-title.txt
    |    |-intro.txt
    |    |-01.txt
    |    |-02.txt
    |    ...
    |    |-reference.txt
    |    |-summary.txt
    ...
    |-back/
```

Note: Where a .txt extension is shown above, the proper extension should be used according to the format indicated in the *Manifest File*. For example `.usfm` or `.md`.

The directories within `content` shown above indicate chapters. There are two special chapters named `front` and `back` that contain, if applicable, the front and back matter.

The files within each chapter represent the chunks of the chapter. Often the chunk file names will be numeric (e.g. `01.txt`) but that is not required. The following reserved chunk names have special meaning:

- `title.txt` - the title of the chapter
- `sub-title.txt` - the sub title of the chapter
- `intro.txt` - the introduction to the chapter
- `reference.txt` - a reference displayed at the end of a chapter
- `summary.txt` - a summary displayed at the end of a chapter

In the case of front and back matter, the above named chunks apply to the project, such as the project title, project summary, etc.

Condensed Form

Note: This specification makes no distinction between condensed and *expanded* forms. This is simply shown as an alternative style. Client applications should be prepared to support your chosen style.

Most *Container Types* support a condensed form in which content in each folder is stored in a single file. e.g.

```
content/
    |-config.yaml
    |-toc.yaml
    |-front/
    |-01/
    |    |-01.txt
    ...
    |-back/
```

Where the file `01.txt` may contain the title, sub-title, intro, chunks etc.

Content Sort Order

When utilizing content in an RC the order is very important. The content sorting rules are defined as:

Chapters

1. front matter directory
2. numeric chapter directories sorted numerically in ascending order
3. non-numeric chapter directories sorted alphabetically
4. back matter directory

Chunks

1. title
2. sub-title
3. intro
4. numeric chunks sorted numerically in ascending order
5. non-numeric chunks sorted alphabetically
6. reference
7. summary

Config

The `config.yaml` file contains information specific to each *RC type*. If a particular *RC type* does not need this file it may be excluded.

Table of Contents

Note: Keys must always be represented even when the value is optional. If the key is not needed/available the value may be empty.

Chapter directories and chunk files are often named with padded integers. A side effect of this is the natural file order often represents the appropriate order. However, you may also indicate the order of chapters and frames by providing a table of contents, `toc.yaml`, within the content directory. If no such file exists then the integer order followed by the natural order of the files will be used.

The table of contents is built with small blocks as shown below. All of the fields in the blocks are optional:

```
---
  title: "My Title"
  sub-title: "My sub-title"
  link: "my-link"
  sections: []
```

The sections field allows you to nest more blocks. The link fields may accept the chunk that should be linked to. Alternatively, you may provide a fully qualified link as defined in *Linking*.

Here is an example `toc.yaml` from translationAcademy. Generally speaking the title and sub-title fields in this file should be the same as what is found in the subsequently named chunks. However, the TOC is allowed to deviate as necessary.

```
---
  title: "translationAcademy Table of Contents"
  sub-title: ""
  link: ""
  sections:
    -
      title: "Introduction to translationAcademy"
      sub-title: "This page answers the question: What is in the Introduction?"
      link: ""
      sections:
        -
          title: "Introduction"
          sub-title: ""
          link: ""
          sections:
            -
              title: ""
              sub-title: ""
              link: "ta-intro"
              sections: []
            -
              title: ""
              sub-title: ""
              link: "uw-intro"
              sections: []
    -
      title: "Table Of Contents - Process Manual Vol 1"
      sub-title: "This page answers the question: What is in the process manual␣
↪volume 1?"
      sections:
        -
          title: "Process Manual Volume 1"
          sub-title: ""
          link: ""
          sections:
            -
              title: "1. Getting Started"
              sub-title: ""
              link: ""
              sections:
                -
                  title: ""
                  sub-title: ""
                  link: "process-manual"
```

```
                sections: []
              -
                title: ""
                sub-title: ""
                link: "getting-started"
                sections: []
          -
            title: "2. Setting Up a Translation Team"
            sub-title: ""
            link: ""
            sections:
              -
                title: ""
                sub-title: ""
                link: "setup-team"
                sections: []
  -
    title: "Table Of Contents - Translation Manual Volume 1"
    sub-title: "This page answers the question: What is in Volume 1 of the␣
↪translation manual?"
    sections: []
```

Alternatively you can choose to use a simplified table of contents as shown below.

Note: We may deprecate this form due to the addition of content sorting instructions describe above. Since sorting is defined this form may not provide anything useful.

```
---
-
    chapter: '01'
    chunks:
        - title
        - '01'
        - '02'
        - '03'
        - '04'
        - '05'
        - '06'
        - '07'
        - '08'
        - '09'
        - '10'
        - '11'
        - '12'
        - '13'
        - '14'
        - '15'
        - '16'
        - reference
-
    chapter: '02'
    chunks:
        - title
        - '01'
        - '02'
        - '03'
```

```
      - '04'
      - '05'
      - '06'
      - '07'
      - '08'
      - '09'
      - '10'
      - '11'
      - '12'
      - reference
```

The simple version will rely on the available content (titles, references, etc.) to generate the table of contents (readable titles will be retrieved from the content itself).

CHAPTER 2

Manifest File

Note: Keys must always be represented even when the value is optional. If the key is not needed/available the value may be empty.

Resource Containers (RCs) have a `manifest.yaml` file that describes its content and structure. Most of the information adheres to the Dublin Core Meta Data Initiative and can be found nested within the `dublin_core` key.

```
---

dublin_core:
    conformsto: 'rc0.2'
    contributor:
        - 'A Contributor'
        - 'Another Contributor'
    creator: 'Wycliffe Associates'
    description: 'The Unlocked Literal Bible is an open-licensed version of the Bible␣
↪that is intended to provide a form-centric translation of the Bible.'
    format: 'text/usfm'
    identifier: 'ulb'
    issued: '2015-12-17'
    language:
        identifier: 'en'
        title: 'English'
        direction: 'ltr'
    modified: '2015-12-22T12:01:30-05:00'
    publisher: 'Door43'
    relation:
        - 'en/udb'
        - 'en/tn'
        - 'en/tq'
        - 'en/tw'
    rights: 'CC BY-SA 4.0'
    source:
        -
```

```
            identifier: 'asv'
            language: 'en'
            version: '1901'
    subject: 'Bible translation'
    title: 'Unlocked Literal Bible'
    type: 'book'
    version: '3'

checking:
    checking_entity:
        - 'Wycliffe Associates'
    checking_level: '3'

projects:
    -
        categories:
            - 'bible-ot'
        identifier: 'gen'
        path: './content'
        sort: 1
        title: 'Genesis'
        versification: 'kjv'
```

Definitions

- `dublin_core`
  - `conformsto`: the version of the RC specification used by the RC.
  - `contributor`: an array of names or aliases to people that have contributed to the resource.
  - `format`: the file format of content within the RC, e.g.
    * `text/usfm`
    * `image/png`
    * `audio/mp3`
  - `identifier`: a *Slug* formatted string uniquely identifying the resource.
  - `issued`: the *Date* of publication.
  - `publisher`: the name of the individual or organization responsible for publishing the resource.
  - `relation`: a array of *Short Links* to related resources.
  - `type`: the RC type.
- `projects`: an array of projects inside the RC.
  - `identifier`: a *Slug* formatted string uniquely identifying the project.
  - `path`: the relative path to the project within the RC. Depending on the RC type this may be a directory or a file.
  - `versification`: the system used for placing verse markers and consequently chunk markers.

Generating From USFM

See *USFM to Manifest* for instructions on populating the manifest.yaml from the headers in a usfm file.

CHAPTER 3

Container Types

Resource Containers (RCs) can be used to represent different forms of data. These different forms are represented by the following types.

The types below are noted by `Type Name (Type Slug)` e.g. `Book (book)`

Note: Most types support a *Condensed Form*.

Book (book)

Represents any text that is structured like a book, there are chapters and chunks.

Config

A book may reference supplementary RCs in it's `config.yaml` file via *Linking*. Such references must be grouped by the corresponding RC type. The order of elements in each of these groups should be respected.

```
---
content:
  01:      // chapter
    01:    // chunk
      dict:
        - '//tw/bible/dict/creation'
        - '//tw/bible/dict/god'
        - '//tw/bible/dict/heaven'
        - '//tw/bible/dict/holyspirit'
      help:
        - '//tq/gen/help/01/01'
        - '//tn/gen/help/01/01'
      img:
        - '//ulb/gen/img/01/01'
```

Note: Implementation Notes: References to external RCs may be displayed in the application along the side of the book content in order to provide contextual information.

Image (img)

Represents a set of images that follows the same structure as a book. In most cases you will want to provide a single image to an equivalent chunk in the related book. However, any image can be included so long as it follows the requirements for identifiers.

Below is an example with a format of `image/png`.

```
content/
    ...
    |-01/
    |    |-title.png
    |    |-sub-title.png
    |    |-intro.png
    |    |-reference.png
    |    |-summary.png
    |    |-01.png
    |    |-02.png
    |    ...
    ...
    |-front/
    |-back/
    ...
```

Audio (audio)

Represents a set of audio files that follows the same structure as a book. It is valid to provide a single audio file to any equilvilant chunk in a book.

Below is an example with a format of `audio/mp3`.

```
content/
    ...
    |-01/
    |    |-title.mp3
    |    |-sub-title.mp3
    |    |-intro.mp3
    |    |-reference.mp3
    |    |-summary.mp3
    |    |-01.mp3
    |    |-02.mp3
    |    ...
    ...
    |-front/
    |-back/
    ...
```

Video (vid)

Represents a set of video files that follows the same structure as a book. It is valid to provide a single video file to any equivalent chunk in a book.

Below is an example with a format of `video/mp4`.

```
content/
    ...
    |-01/
    |    |-title.mp4
    |    |-sub-title.mp4
    |    |-intro.mp4
    |    |-reference.mp4
    |    |-summary.mp4
    |    |-01.mp4
    |    |-02.mp4
    |    ...
    ...
    |-front/
    |-back/
    ...
```

Help (help)

Note: These types do not support a *Condensed Form*.

A helpful resource to supplement chunks in a book. e.g. notes or questions. Currently all help RCs must use the markdown format.

Each chunk contains one or more helps which correlate to the corresponding chunk in a book RC:

```
# In the beginning God created

This introductory statement gives a summary of the rest of the chapter. AT: "This is␣
↪about how God made...in the beginning." Some languages translate it as "A very long␣
↪time ago God created." Translate it in a way that that shows that this actually␣
↪happened and is not just a folk story.

# In the beginning

This refers to the start of the world and everything in it.
```

When parsed by an app the helps in this chunk are split at the headers. If there is preceding text (without a header) it will be displayed as a single help and a short snippet of the text will be used for the header if applicable.

Dictionary (dict)

A standalone dictionary of terms. Currently all dictionary RCs must use the markdown format.

The dictionary terms are used as the chapter *Slug* and the description of the term is placed inside a `01.txt` file:

```
content/
    |-config.yaml
    |-aaron/
    |    |-01.txt
    |
    |-abel/
    ...
    |-unclean/
```

Note: Lengthy dictionary terms may be split into more than one chunk.

The `01.txt` file contains the description of the term where the header is the title of the term and the rest is the description:

```
# Aaron

God chose Aaron to be the first high priest for the people of Israel.
```

The `config.yaml` file is used to indicate related terms, aliases, definition title, and examples.

```
---
  aaron:
    def_title: 'Description'
    see_also:
      - 'covenant'
      - 'testimony'
    aliases:
      - aaronalias # note: not a real alias for this word
    examples:
      - '09-15'
      - '10-05'
```

Examples are tricky because a `dict` may be referenced by many different resources and projects. Therefore we cannot specify a RC link but instead must simply provide the chapter and chunk that contains the example.

Manual (man)

A user manual. For now manual RCs must use the markdown format.

Manuals are a collection of modules (articles):

```
content/
    ...
    |-translate-unknowns
    |    |-title.txt
    |    |-sub-title.txt
    |    |-01.txt
    ...
    |-writing-decisions/
```

The `01.txt` file contains the translation of the module.

Note: If desired the module can be split into multiple chunks.

The `config.yaml` file indicates recommended and dependent modules:

```
---
  translate-unknowns:
    recommended:
      - 'translate-names'
      - 'translate-transliterate'
    dependencies:
      - 'figs-sentences'
```

Dependencies are *Slug* s of modules that should be read before this one. Recommendations are modules that would likely benefit the reader next.

Bundle (bundle)

A bundle is simply a flat directory (no sub-folders) with a single file for each project. This type is particularly suited for USFM when providing "USFM Bundles".

A project block in the *Manifest File*:

```
---
  projects:
    -
      identifier: 'gen'
      title: 'Genesis'
      versification: 'kjv'
      sort: 1
      path: './01-GEN.usfm'
      categories:
      - 'bible-ot'
```

Directory structure

```
myrc/
    |-01-GEN.usfm
    |-LICENSE.md
    |-manifest.yaml
```

Note: When your application supports "USFM Bundles" it can identify the them in two ways

- attempt to read the *Manifest File* to determine type as `bundle` and the format as `text/usfm`.
- look for any `*.usfm` files in the root directory if the *Manifest File* does not exist.

In this way the application will satisfy both the `Bundle` RC type described above and generic "USFM Bundles" as is common in the industry.

CHAPTER 4

Linking

A Resource Container (RC) link allows one RC to reference content from another RC.

All RC links follow a very simple structure in two different flavors.

- **Anonymous links** - have no title and are declared by enclosing the link in double brackets
- **Titled links** - have a title and are indicated by enclosing the link title in single brackets and the link in parentheses.

For example:

```
[[language/resource/project/type]]

[Link Title](language/resource/project/type)
```

Structure

The minimum form of a link is `language/resource/project/type`. We interpret this as the project content directory inside the RC. This is illustrated below:

```
# link
en/ulb/gen/book

# file system
en_ulb_gen_book/
    |-LICENSE.md
    |-manifest.yaml
    |-content/ <-- link points here
```

From this point we can lengthen the link to include a chapter *Slug* which resolves to the chapter directory.

```
# link
en/ulb/gen/book/01
```

```
# file system
en_ulb_gen_book/
    |-LICENSE.md
    |-manifest.yaml
    |-content/
        |-01/ <-- link points here
```

Going a step further we can link to a specific chunk

```
# link
en/ulb/gen/book/01/01

# file system
en_ulb_gen_book/
    |-LICENSE.md
    |-manifest.yaml
    |-content/
        |-01/
            |-01.usfm <-- link points here
```

In some of the examples above the link was not pointing directly at a file. In those cases the link should resolve to the first available file in order of the sorting priority described in *Content Sort Order*.

External URLS

You may link to online media by simply using a url instead of an RC identifier.

- `[[https://www.google.com]]`
- `[Google](https://www.google.com)`

Links where the path begins with `http://` or `https://` are treated as external urls.

Examples

book

- `[Genesis 1:2](en/ulb/gen/book/01/02)`
- `[Open Bible Stories 1:2](/en/obs/obs/book/01/02)`

help

- `[[en/tq/gen/help/01/02]]` - links to translationQuestions for Genesis 1:2
- `[[en/tn/gen/help/01/02]]` - links to translationNotes for Genesis 1:2

dict

- `[Canaan](en/tw/bible/dict/canaan)`

man

- `[Translate Unknowns](en/ta-vol1/translate/man/translate-unknowns)`

img

- `[Open Bible Stories 1:2](en/obs/obs/img/01/02)`
- `[Genesus 1:2-6](en/ulb/gen/img/01/02)`

vid

- `[Open Bible Stories 1:2](en/obs/obs/vid/01/02)`

audio

- `[Open Bible Stories 1:2](en/obs/obs/audio/01/02)`

bundle

- `[Genesis](en/ulb/gen/bundle/01/01)`

Note: Linking to a bundle will only resolve down to the project level. e.g. the `01/01` will be ignored and the entire project returned. If you must link to a section within the project you will have to parse the content and manually resolve the rest of the link if the `format` supports references.

Formats that support references are:

- usfm
- osis

Abbreviations

In certain cases it is appropriate to abbreviate a link. Below are a list of cases where you are allowed to use an abbreviation.

Links within the same RC

When linking to a different section within the same RC you may just provide the chapter/chunk *Slug* s.

Manual example:

- `[Translate Unknowns](translate-unknowns)`

Dictionary example:

- `[Canaan](canaan)`

Book example:

- `[Genesis 1:2](01/02)`

Links to any language

At times you may not wish to restrict the link to a particular language of the RC. In that case you may exclude the language code from the beginning of the path and place an extra slash `/` in it's place.

Example:

- `[[//ta-vol1/translate/man/translate-unknowns]]`
- `[Translate Unknowns](//ta-vol1/translate/man/translate-unknowns)`

Short Links

A short link is used to reference a resource but not a project. There is nothing fundamentally different from regular links. Short links are simply composed of just the language and resource.

- `en/tn`

Short links are most often used within the *Manifest File* when referring to related resources.

Automatically Linking Bible References

Bible references in any RC should be automatically converted into resolvable links according to the linking rules for **book** resource types. Of course, if the reference is already a link nothing needs to be done.

Conversion of biblical references are limited to those resources that have been indexed on the users' device. Conversion should be performed based on any one of the following:

- a case *insensitive* match of the entire project title.
- a start case (first letter is uppercase) match of the project *Slug* e.g. `Gen`.

For each case above there must be a valid `chapter:verse` reference immediately after the matching word separated a single white space. For example:

```
Genesis 1:1
genesis 1:1
Gen 1:1
Gen 1:1-3
```

The chapter and verse numbers should be converted to properly formatted *Slug* s.

Example

Given the French reference below:

`Genèse 1:1`

If the user has only downloaded the English resource the link will not resolve because the title `Genesis` or `genesis` does not match `Genèse` or `genèse`. Neither does the camel case *Slug* `Gen` match since it does not match the *entire* word.

If the user now downloads the French resource the link will resolve because `Genèse` or `genèse` does indeed match `Genèse` or `genèse`. The result will be:

```
[Genèse 1:1](fr/ulb/gen/book/01/01)
```

Multiple Matches

When a match occurs there may be several different resources that could be used in the link such as `ulb` or `udb`. When more than one resource *Slug* is available use the following rules in order until a unique match is found:

1. use the same resource as indicated by the application context.
2. use the RC allowed by the translate_mode set in the application.
3. choose the first resource found or let the user choose (e.g. pop up).

Aligning Verses to Chunks

Because chunks may contain a range of verses, a passage reference may not exactly match up to a chunk. Therefore some interpolation may be nessesary. For both chapter and verse numbers perform the follow:

> Given a chapter or verse number `key`. And an equivalent sorted list `list` of chapters or verses in the matched resource

- incrementally compare the key against items in the list.
- if the integer value of the current list item is less than the key: continue.
- if the integer value of the current list item is greater than the key: use the previous list item.
- if the end of the list is reached: use the previous list item.

For example chunk `01` may contain verses `1-3` whereas chunk `02` contains verses `4-6`. Therefore, verse `2` would resolve to chunk `01`.

If no chapter or chunk can be found to satisfy the reference it should not be converted to a link.

CHAPTER 5

App Meta

Applications can store meta data in a Resource Container (RC) within the `.apps` directory.

Note: The contents of this directory are outside the scope of the RC specification. The examples below are only a suggestion.

translationStudio

translationStudio keeps track of stages chunks go through during translation.

```
my_rc/.apps/ts/
    |-status.json
```

translationCore

translationCore keeps track of checking information.

```
my_rc/.apps/tc/
    |-checkdata/
    |   |-LexicalChecker.tc
    |   |-ProposedChanges.tc
    |   |-common.tc
    |
    |-tc-manifest.json
```

CHAPTER 6

USFM to Manifest

When converting a USFM file such as `01-GEN.usfm` into an RC follow the rules below when populating the *Manifest File*.

Map

Some values are known by default since USFM is always used for Bible projects:

- *dublin_core:type* = `book`
- *dublin_core:conformsto* = `rc0.2`
- *dublin_core:format* = `text/usfm`
- *dublin_core:subject* = `Bible translation`
- *projects:categories* = `bible-ot` or `bible-nt` depending on the project identifier

The rest may be parsed from the USFM:

id_<CODE>_(Name of file, Book name, Language, Last edited)

- `code` -> *projects:identifier*
- `Book name` -> *dublin_core:title*
- `Language` -> *dublin_core:language:title*
- `Last edited` -> *dublin_core:modified*

h_text...

- `text` -> *projects:title*

The following items will need to gathered by the person or app doing the conversion:

- *dublin_core:identifier*
- *dublin_core:language:identifier*

- *dublin_core:language:direction*
- *dublin_core:rights*
- *dublin_core:contributor*
- *projects:versification*

CHAPTER 7

Slug

A slug is a word or a few words that uniquely describe an object. Slugs are often used instead of an id (from a database) due to their readability and portability.

Syntax

Slugs are composed of lowercase alphanumeric characters and hyphens.

```
abcdefghijklmnopqrstuvwxyz1234567890-
```

- The first character in a slug *must* be a letter.
- The last character in a slug *must not* be a hyphen.

Examples

```
gen
pt-br
aey-x-haya
custom-slug-123
my-1st-slug
```

CHAPTER 8

Date

Dates follow the encoding scheme defined in the W3CDTF profile of ISO 8601.

Example

```
1994-11-05
1994-11-05T08:15:30-05:00
```

Corresponds to:

```
November 5, 1994
November 5, 1994, 8:15:30 am, US Eastern Standard Time.
```

CHAPTER 9

Code Libraries

Below are a few libraries designed to work with Resource Containers (RCs). If you would like to have your library displayed in this list please open an issue on GitHub.

Android

- unfoldingWord-dev/android-resource-container

Node.js

- unfoldingWord-dev/node-resource-container

Python

- unfoldingWord-dev/python-resource-container