LiquiDoc Documentarian Manual

toc::[]

Table of Contents

The latest and greatest version of this manual can be found in many formats at {this_guide_base_url}.

Welcome to the LiquiDoc Documentarian Manual!

Here you will learn to write, edit, and manage great documentation for LiquiDoc and LiquiDoc CMF as a documentarian.

This Manual is intended to orient you to the role of **documentarian**, a LiquiDoc user mainly concerned with *creating and managing content*. This Manual expects that you have a pre-configured LiquiDoc CMF environment established by a qualified engineer, and you just need to get up and running so you can write/edit content. Hopefully, whoever did that configuring has edited this documentation in key areas to ensure it applies to your instance.

In this guide, you will learn to…

- grasp the significance of LDCMF and its components
- get to know the codebase and perform an initial build
- establish a strong local toolset for working in LiquiDoc
- establish a complete toolset on Windows
- grasp the role of LDCMF documentarian
- get familiar with this LDCMF environment
- grasp the differences between content and data formats
- write and edit content with AsciiDoc
- write and edit small data with YAML
- employ token substitution with Liquid variables and AsciiDoc attributes
- locate and manage source files
- use cross-reference shortcut attributes in AsciiDoc
- commit, submit, and iterate source matter for publication
- Review and approve data/content commits for publication
- follow clear steps to get unstuck when problems arise
- understand my rights as a contributor
- understand how merge requests are evaluated

This document contains lots of jargon. See the AJYL-centric DocOps Jargon Glossary if you get lost.

Getting Comfortable with LiquiDoc CMF

LiquiDoc CMF (LDCMF) is a "content management *framework*" — a set of tools, conventions, and standards for sensibly organizing and maintaining source content and data as well as producing deliverable artifacts. The LiquiDoc/LDCMF User Guides project covers the general use and maintenance of LiquiDoc and LiquiDoc CMF. Here we'll preview its main features, focusing on how they're implemented in this project (LiquiDoc/LDCMF User Guides).

LDCMF: Not Your Granddad's Content Platform

LDCMF is likely to seem unfamiliar. It is a publishing platform but not a content management *system* (CMS) nor a *component* content management system (CCMS). LDCMF differs from these mainly in that it does not revolve around a database or a user interface designed for a specific type of content or publishing.

LDCMF is designed for flexibility, in order to meet the demands of complex documentation projects that cover potentially numerous *versions* of multiple *products* for various *audiences*, perhaps yielding artifacts in two or more output *formats*, as well as other complicating factors. The platform enables a docs-as-code approach to technical content, whereby the documentation source material is tied to the product source code, as well as using tools and methods more familiar to developers than to writers.

Pros/Cons of LiquiDoc CMF vs Proprietary CMS/CCMS Solutions

There are several major differences between an open-source docs-as-code approach to creating, managing, and publishing technical documentation. Whether they are *pros* or *cons* in your book may depend very much on whether your background.

Some assembly required

Users are expected to heavily customize and extend their LDCMF environment rather than fall back on "turnkey" features and elements. While you can technically write and build a pretty straightforward docset based on existing, freely available LDCMF examples, your needs will almost certainly vary. The free-and-open model adhered to by the *framework* means you will never encounter a dead end imposed by the LDCMF platform. Hopefully you won't need to actually *modify or extend* the base tooling to solve your needs, but you will need to *configure* a complex docs build if you have complex problems. Presumably, that's how you ended up here anyway.

Small data is simple

LDCMF's sources of data and content are far simpler than conventional CMS applications. Stored in flat files and directories rather than relational databases (RDBs), LDCMF's relatively flexible and casual datasource options have their limitations. Most conventional CMS platforms take advantage of RDBs' powerful indexing and querying capabilities, not only handling *content and data* but managing large amounts of site and content *metadata*. On the other hand, RDBs cannot be used with distributed source-control systems like Git.

GUI? What GUI?

LiquiDoc CMF's user interface is command-line tools (CLIs) and free, open-source text/code editors, rather than proprietary desktop programs or web apps. For some, this is the epitome of power and freedom. For others, these blinking-cursor options are intimidating to the point of paralysis. While LDCMF can be deftly operated by *beginners* with both kinds of tools, there may be some initial discomfort. But then: *total freedom and power!*

LDCMF is AJYL

The core component technologies of the LDCMF platform are Asciidoctor, Jekyll, YAML, and Liquid—open-source platforms and formats that in combination make enterprise-scale single-source publishing possible. Together, these packages form the AJYL docstack, a robust documentation ecosystem with the accessibility, flexibility, and compatibility needed for confident, open-ended development. LiquiDoc is simply a utility for tying these technologies together, while LDCMF is a set of conventions and strategies for building great docsets from canonical sources managed in Git. For more, check out the AJYL landing page.

What Do I Need to Learn?

First, be sure you're looking at the guide for the proper role (Documentarian). See Understanding Your Role as Documentarian to be sure.

In your role as a documentation contributor, or *documentarian*, you will not need to know the differences between LiquiDoc, LDCMF, and your team's implementation, so this overview should suffice. Instead, this Manual will walk you through the procedures necessary to create, edit, and manage content in this particular LDCMF environment.

LiquiDoc and LDCMF Overview

The LiquiDoc CMF platform relies on the LiquiDoc build utility, which in turn employs other open-source applications to process and render rich-text and multimedia documentation output.

As should be clear from the comparisons key to LDCMF-based documentation projects is managing all content and data in plaintext ("flat") files rather than a database. The primary source formats for an LDCMF project like this one (LiquiDoc/LDCMF User Guides) are **AsciiDoc** and **YAML**.

AsciiDoc

: Dynamic, lightweight markup language for developing rich, complex documentation projects in flat files. (Resource)

YAML

: A slightly dynamic, semi-structured data format for key-value pairs and nested objects (Resource)

These formats are chosen for efficacy, learnability, and readability, and this guide will walk you through the steps you need to get comfortable and proficient with them, including plenty of supplemental resources. Before diving into AsciiDoc and YAML, let's keep exploring just what LDCMF is.

LiquiDoc CMF is used to build various types of documentation, but it excels at multi-format output, such as generating a PDF edition and a website *from the same source files*.

The Intimidation Factor

While LDCMF takes advantage of developer-oriented utilities and procedures, it is designed for tech-savvy content professionals who want to work more closely to the code *and the engineering team*.

Instead of web forms with text fields, selectboxes, and WYSIWYG editors; LDCMF offers a bunch of text files. The advantages may not be self-evident yet, but let's address the elephant in the room: this all seems a lot harder than it should be. It will sometimes take more work to manage docs in plaintext files using what will at first feel like crude editing tools, not to mention that clumsy command line.

While LDCMF's usability will steadily improve, it will always require technical writers and documentation managers who have worked in other fields to reconceive how docs are created and managed. But you *will* be able to get comfortable with your new tooling, and you might even come to appreciate it. Nothing we can say here will take the pain away, but rest assured this documentation is written with beginners in mind.

GitHub

This project is intended to be managed using **GitHub**, the most popular cloud service for Git repository hosting. You will need a GitHub account to fully participate as a contributor.

Unfortunately, GitHub's friendly user interface will only handle some of the procedures you need to perform in order to commit to documentation.

Managing content directly with Git allows documentation to more accurately align with the product it covers, a key objective of LDCMF.

If you're a technical writer who has avoided Git so far, your number may be up. There's great news: Git is a highly marketable skill with an ever-widening range of applications in various sectors, fields, and roles. Once you experience its potential in the world of documentation, you will understand what everyone is so excited about.

Getting Started with the Documentation Codebase

You can get started with the LiquiDoc CMF environment used to document **LiquiDoc and LiquiDoc CMF** right away. This orientation will introduce you to the LiquiDoc/LDCMF *docs* codebase as well as its tooling, conventions, and workflow.

You are currently reading the docs for managing our docs for AJYL docstack, LiquiDoc, and LiquiDoc CMF (LDCMF). These guides instruct proper use of our LDCMF implementation, including how to contribute to and manage our product docs, as well as how to administer the LDCMF instance.

Your Role

As a *documentarian*, your environment should already be defined, though you may still need to set it up on your workstation. The contents of this Manual are customized for your team's needs.

Once we make sure you have the few prerequisites in place, you can build these docs. That will be essentially the entire operation for documentarians. You will then be able to start creating or editing content, then rebuilding to see your work in place.

For more on your role as Documentarian, see the

Installing Dependencies

This procedure invokes the LiquiDoc tool and in turn Asciidoctor, Jekyll, and other dependencies to build all documentation and other artifacts.

The only prerequisite software packages you may need are Ruby runtime and Git. You will also need a terminal application. The rest will be installed during the basic setup instructions.

Terminal

If you are a Linux user, hopefully you already know your favorite terminal. For **MacOS**, use spotlight to find your `terminal`, or try **iTerm2**.

Windows users should use the **GitBash** terminal installed the next step.

Git

If you are just getting started with Git, this GitHub resource guide may have something for your learning style.

For setting up **Git on Windows**, use the GitForWindows guide.

Ruby Runtime Environment

If you're on **Linux or MacOS**, you probably have Ruby runtime. Using your preferred terminal application, check your Ruby version with `ruby -v`.

If you're on Windows, use this download page. Ruby version must be 2.3.0 or higher; 2.5+ recommended, development kit not required.

Anne Gentle has provided excellent instructions for getting up and running on Windows with Ruby and Jekyll. (There are some good **MacOS** tips there as well.)

Quickstart

Open your preferred terminal, and navigate to a workspace in which you can create a new subdirectory for the local repository ("repo").

1. Clone this repo.

   ```
   git clone git@github.com:DocOps/liquidoc-cmf-guides.git liquidoc-cmf-guides
   ```

   Now you have a local copy of the repository files.

2. Change your working directory to the docs directory.

   Example

   ```
   cd liquidoc-cmf-guides/
   ```

3. Run Bundler to install dependencies.

   ```
   bundle install
   ```

   If Ruby says you don't have Bundler installed, run `gem install bundler`.

4. Run your first build of these docs.

   ```
   bundle exec liquidoc -c _configs/build-docs.yml
   ```

   This executes a specific build routine using the LiquiDoc utility through Bundler, basing the build procedure on a config file.

5. Serve the website locally.

   ```
   bundle exec jekyll serve --destination build/site \
     --config _configs/jekyll-global.yml --skip-initial-build --no-watch
   ```

Now you're able to view the LiquiDoc/LDCMF User Guides web portals and associated artifacts, right on your local machine. Browse http://127.0.0.1:{local_serve_port}.

Full Command

Use this command to execute a clean, build, and serve operation locally.

```
rm -rf _build && bundle exec liquidoc -c _configs/build-docs.yml && bundle exec jekyll
serve --destination _build/site --config _configs/jekyll-global.yml --skip-initial
-build
```

Special Build Options

Here are some special flags that work with this project's primary build config (`_configs/build-docs.yml`).

Build without Asciidoctor rendering (Jekyll or PDF)

```
bundle exec liquidoc -c _configs/build-docs.yml -v norender=true
```

Build without rendering website files

```
bundle exec liquidoc -c _configs/build-docs.yml -v nojekyll=true
```

Build without rendering PDFs

```
bundle exec liquidoc -c _configs/build-docs.yml -v nopdf=true
```

What Just Happened?

The only steps you'll need to perform regularly going forward will be the last two. But all these steps are relevant to your work, so we'll exlore them one by one.

Ruby Runtime

Whether you already had a Ruby runtime environment or just installed it, you're now able to execute packaged Ruby scripts called "gems".

Unless you intend to modify (hack) LiquiDoc, Asciidoctor, or Jekyll yourself, you don't need to know anything about the Ruby language to use these utilities. However, it is handy to understand a little about how Ruby works on your system and how you will be engaging with it. That orientation starts below with Bundler, but first we should set the stage some more.

Git

If you were not familiar with Git before, you are about to become intimate. We'll be exploring Git operations in the LiquiDoc CMF Overview. For now, the relevant aspect of Git is that you have

created a local Git repository during the first step above. This step executed a Git command (`git clone`) to grab a copy of this repo from the remote address and clone it to your system. In so doing, it initialized that root directory as a Git repository—not just any set of files. This means your repository is ready for action, and all the powers of Git are at your fingertips. You'll be using more of them soon enough.

Project Working Directory

Every LiquiDoc CMF project has a base directory from which it is best to run all commands. Always navigate into this directory when you begin working on content, so any `liquidoc`, `asciidoctor`, or `jekyll` commands you may find in these instructions are always run from that base.

If you ever need to know what directory you are in, enter `pwd` at the prompt and the full path will display.

Bundler

The first Ruby "trick" you should be familiar with is `bundle`, the command that invokes the Bundler gem. For our purposes, Bundler reads the file simply called `Gemfile`, which you will find in your project root directory. Bundler gathers packages, primarily from Rubgems.org, an open-source gem package hosting service. This `Gemfile` defines dependencies used by LiquiDoc, Jekyll, and Asciidoctor as they process source code into publications during a build procedure. *Engaging Bundler during every execution of these key Ruby gems ensures proper versions of all their prerequisites are in order.*

Running `bundle update` on the command prompt will always check for and install the latest gem updates, which should be pretty safe to do from time to time.

LiquiDoc build procedure

The `bundle exec liquidoc` command executes the utility that coordinates the complex documentation build procedure. This is all instructed in the build-configuration file indicated in the command (`_configs/build-docs.yml`). We'll explore that file and the entire build operation much further in the LiquiDoc CMF Overview.

For now, it is most helpful to understand the role LiquiDoc plays in the build process, which is mainly just the orderly invocation of more-powerful tools like Liquid, Asciidoctor, and Jekyll. The `liquidoc` command accepts a range of options, but you'll be running it under fairly strict instructions.

At the end of this procedure, we have generated PDF artifacts as well as static HTML files completely parsed, compiled and ready to serve. You can always exclude either the PDF artifacts or the Jekyll portals from the build.

```
bundle exec liquidoc -c _configs/build-docs.yml -v nopdf=true
```

To skip the PDF build, use `-v nopdf=true`. To skip the Jekyll build, use `-v nojekyll=true`.

Jekyll Serve Procedure

This step fires up a local “development server”, giving us a proper browser protocol for navigating all those HTML files. We look more deeply at the role Jekyll plays in LiquiDoc CMF in LiquiDoc CMF Overview.

This specific `jekyll serve` command was run with some special options. Without delving into too much detail, these options serve all the pages we want at once, for multiple portals, and disables default Jekyll features that would interfere with our operations.

The reason we have to run this step separately is that the build we performed in the last step created multiple Jekyll sites (our “portals”), and we have to serve Jekyll with specific commands in order to deploy the artifacts together. This step will be integrated into the LiquiDoc configuration when LiquiDoc is better able to accommodate complex Jekyll commands.

Establishing a Local Toolset

Running LiquiDoc on Windows

Understanding Your Role as Documentarian

As reflected in the two separate guides this repository generates, there are two fairly distinct user-persona categories expected to engage with the documented LiquiDoc/LDCMF instance.

The Roles

Anyone producing or editing *content* for this project — whether their title is *engineer*, *technical writer*, or *documentation manager* — is acting in the role of ***documentarian***: a documentation *contributor*. You are currently reading the Documentarian Manual.

Another key role is that of ***administrator***, or ***admin***. This role is responsible for configuration and management of all the technologies underlying the docs build, as well as overseeing adherence to conventions and best practices by documentarians. Anyone fulfilling this role must be comfortable with all of the information in both the Documentarian and Administrator Guide. You are currently reading the Documentarian Manual.

The (Documentarian) Role

A documentarian contributes to the LiquiDoc/LDCMF User Guides codebase, either as a LiquiDoc and LiquiDoc CMF developer reporting new functionality or information, or as a technical writer whose main role is to improve the LiquiDoc/LDCMF docs.

Documentarians generally need to be familiar with AsciiDoc and Liquid markup, as well as the basic ways the files in the docs repository relate to one another. Additionally, documentarians should be able to build the output artifacts locally and know how to troubleshoot various problems their changes might introduce.

Getting to Know This LDCMF Environment

The LDCMF Guides project is itself a complex implementation of the LiquiDoc Content Management Framework, so it serves as a good example via which we can examine some of LDCMF's various features.

A Quick Review

The files in this repository are *written* and *edited* in the AsciiDoc and YAML lightweight markup formats, using your code editor of choice. Then they are compiled into rich-text output ("artifacts") by LiquiDoc during the *build* procedure. During this build, LiquiDoc engages Liquid (template parsing engine), Asciidoctor (AsciiDoc rendering engine), and Jekyll (static-site generator) to generate HTML files and build them into a configurable array of pages for publication. The text files comprising the source content are *managed* using Git.

The rest of this topic breaks that process down in some detail, but here is a bit more orientation.

The end products of this source code are a website containing multiple "portals"--one for each user role, the broad personas expected to engage with LDCMF. Those portals share a tremendous amount of content in common, but they vary from one another in a number of significant ways. Therefore, their source matter is stored predominantly in common files, differentiated where the products diverge, then processed into separate, collocated sites.

Repo Tour

Review this partial exposure of the standard LDCMF directory tree.

Basic LDCMF File Structure

```
ldcmf-guides/
 ├──── _build/
 ├──── _configs/
 │     ├──── asciidoctor.yml
 │     ├──── build-docs.yml
 │     ├──── jekyll-global.yml
 │     ├──── jekyll-guide-admin.yml
 │     ├──── jekyll-guide-dev.yml
 │     ├──── jekyll-guide-docpro.yml
 │     └──── jekyll-guides-common.yml
 ├──── _ops/
 ├──── _templates/
 │     └──── liquid/
 ├──── content/
 │     ├──── assets/
 │     │     └──── images/
 │     ├──── snippets/
 │     ├──── pages/
 │     ├──── special/
 │     │     └──── assets/
 │     │          ├──── css/
 │     │          ├──── fonts/
 │     │          ├──── images/
 │     │          └──── js/
 │     ├──── topics/
 │     └──── guides-index.adoc
 ├──── data/
 │     ├──── guides.yml
 │     ├──── meta.yml
 │     ├──── products.yml
 │     ├──── schema.yml
 │     └──── terms.yml
 ├──── products/
 │     ├──── cmf/
 │     └──── gem/
 ├──── theme/
 │     ├──── css/
 │     ├──── docutheme/
 │     │     ├──── _includes/
 │     │     └──── _layouts/
 │     ├──── fonts/
 │     └──── js/
 ├──── Gemfile
 ├──── Gemfile.lock
 ├──── NOTICE
 └──── README.adoc
```

Now let's dig into the particulars.

_build/

This is where all processed files end up, whether we're talking migrated assets, prebuilt source, or final artifacts. This directory is *not* tracked in source control, so you will not see it until you run a build routine, and you cannot commit changes made to it. It is always safe to fully delete this directory in your local workspace. We will explore the `_build/` directory more fully later.

_ops/

This is a secondary "configs" location, for utilities and routines that support the *use* of LDCMF by admins and documentarians. For instance, the `init-topic.yml` config instructs the creation of topic files and schema entries.

_templates/liquid/

Here we store most of our prebuilding templates. These are *not* Jekyll theming templates. These are the ones we use for generating new YAML and AsciiDoc source files from other source files and external data.

content/

The first of our publishable directories, `content/` is the base path for documentarians' main work area. Everything inside the `content/` directory will be copied into the `_build/` directory early in the build process.

content/assets/

For content assets, rather than theming assets. If it illustrates your product, it probably goes here. If it brands your company, it probably goes in `theme/assets/`.

content/pages/

For AsciiDoc files of the *page* content type. See [{XREF_source-content-types}].

content/snippets/

For content *snippets*. See [snippets].

content/topics/

For AsciiDoc files of the core *topic* content type.

data/

All YAML small-data files that contain content-relevant information go here. These data files differ from those that belong in `_configs/` (or `_ops/`) in important ways, essentially revolving around whether the data needs to be available for display. If it is not establishing settings or used to inform non-build functions (like in `_ops/`), the data file probably belongs in `data/`. Let's look at some key data files standard to LDCMF.

data/meta.yml

For general information about your company, URL and path info. This file usually contains just simple data: a big (or small) column of basic key-value pairs to create simple variables.

data/products.yrml

For subdivided information about your products in distinct blocks. Each block can be called for selective ingest during build routines using the colon signifier, such as by calling

`data/products.yml:product-1`, where `product-1:` is a top-level block in the `products.yml` file.

data/guides.yml

This block is for content-oriented data that is distinct between the different portals or guides you're producing. This is often redundant to your `products.yml` file, if product editions themselves are the major point of divergence in your docs, and it is formatted the same way. For *this project* (LDCMF Guides), the *guides* are oriented toward *audiences* (documentarians, admins, and developers), but the products (LiquiDoc and LDCMF) are distinct from this and actually documented/instructed *together* in each guide.

Favoring the filename `products.yml` is conventional when products and guides (portals) have a 1:1 relationship and `guides.yml` file is superfluous.

data/schema.yml

Can also be `data/manifest.yml`, this crucial file provides, central manifest of all page-forming content items (pages, topics) and how they are organized (metadata such as categories into which content items fall). The schema file carries essential build info that lets us see relationships between topics and build content-exclusive portals from otherwise fairly dumb, decontextualized repositories.

data/terms.yml

By no means a required file, `terms.yml` is a great example of a file that is really just for content. You can have as many of these key-value files, serving whatever purposes you may wish.

products/

This is an optional path for LDCMF projects. If you plan to embed your product repos as submodules, the `products/` directory is the base path to stick them in. For LDCMF Guides, this path effectively leads to symlinks for the LiquiDoc and LDCMF repos, so any files within those repos are accessible to be drawn into our docs.

theme/

All the files that structure your output displays go here. This mainly includes Jekyll templates (`themes/<theme-name>/_includes.yml` and `themes/<theme-name>/_layouts.yml`) and asset files such as stylesheets, front-end javascripts, and of course theme-related images. This would also be the home of PDF and slideshow output theme configurations, as applicable.

Markup Basics

This project is made up of structured information that comes in two forms: **content** and **small data**. Each of these forms of info is stored (sourced) in a respective lightweight markup format: **AsciiDoc** (for content) and **YAML** (for small data). Along the way to being built into richtext documents, data markup and content markup are mixed using **Liquid** templates, generating still more AsciiDoc-formatted source files for the final build.

Basic familiarity with AsciiDoc and YAML is required is required, but most of the work contributors perform will be carried out using a range of markup elements that can be learned very quickly. Most people find basic AsciiDoc as easy to learn and use as Markdown, and YAML is generally considered more human-friendly than JSON. Liquid is also just a markup format, but it has still more elements of a programming language, such as iteration, text manipulation, and advanced conditionals. With increased complexity comes tremendous power to reshape documents from any plaintext format to any other plaintext format.

So let's start with the *what* and *why* before moving on to the *how*.

Content: AsciiDoc Basics

Documentarians will likely spend most of their *writing* time in AsciiDoc files. Whether you're an SME or a technical writer, *content* is where *data* is contextualized and explained for the end user. So while you will have data at your disposal during your writing (in the form of variables), you will actually be somewhat removed from the substance of that data until you perform a build operation and check it in its rendered context. That's when conditionals and variables get evaluated and output gets real.

Yet content is not just the words that connect data, either. Content includes an internal *structure* that gives *semantic relevance* to every element in the final output. **Semantic structure** includes headings; block names in addition to section titles; admonition blocks like tips, warnings, and notes; code and shell-command listings; examples; sidebars; definition lists; and numerous other types of elements. These elements can be sourced as plaintext but rendered in various rich-text display formatting to denote different purpose, importance, or relevance. For more on the subject of semantics in technical writing: Semantic Meaning in Authoring Documentation.

Elsewhere we explore the nitty gritty of Writing and Editing Content with AsciiDoc.

Small Data: YAML Basics

AsciiDoc is to *written content* as YAML is to *structured data*. Whenever possible, we prefer YAML for **small data** because it's easy and elegant, unlike JSON and XML, though we can fall back to these formats as needed. Additionally, we may use CSV format, since it is easy to manage in collaborative tooling. It may not be obvious what should be stored in data files, versus what should be written directly in AsciiDoc files.

A piece of source matter qualifies as *small data* that should be handled in a YAML file if it meets *both* of the following conditions:

1. It will appear in more than one place in the docs.
2. It can be expressed as a simple string or integer.

We will address how we decide *where* to store such data shortly.

The first type of structured information that qualifies as small data is **simple key-value pairs**, which is trivial to identify. Here is an example of simple variables that qualify under the above conditions:

Simple variables in YAML

```
support_phone_number: 888-923-1342
company_site_url: https://www.example.com/
comany_site_domain: example.com
```

If you find yourself typing any specific piece of information that needs to be exact but that you might not want to memorize, you have a candidate for small data that should be stored in a file in the `data/` directory.

A piece of content that will appear in more than one place but is *not* suitably expressed as a simple string or numeral is most likely a *snippet*, which is handled differently (see [{XREF_content-snippets}]).

If you are a documentarian, you can skip to [what-goes-where].

The above simple variables are globally applicable throughout a document and can be called in AsciiDoc or Liquid like so:

- `{company_name}` (AsciiDoc)
- `{{ company_name }}` (Liquid)

Use of variables is covered more fully in Working with AsciiDoc Attributes and Liquid Variables.

There are other conditions under which more complex forms of data might be suitable for handling as small-data in YAML files. This is the kind of data ***administrators*** are responsible for setting up ahead of time.

highly discrete & simple

Here we mean each datum is a small, contained piece of information, lightly formatted with markup if at all. Code samples longer than a line would not qualify, nor would multi-paragraph content.

restricted scale

Nobody wants to manage more than a few hundred records in a flat-file format like YAML. If your serialized data will eventually contain thousands of currently valid entries, it's time to look into an engine-backed database.

minimal relationships

Unlike relational databases that can be queried using SQL, YAML has no concepts such as *tables*, and no way to relate them like unions and joins. If your data relies on external references, it's database time. YAML does allow nesting, however, so simple would-be cross references can be accommodated.

consistently serialized

If the content is part of a series of repeated items with similar makeup, it can be better represented in YAML than AsciiDoc. The terms in a glossary are serialized as multiple entries that are siblings to one another, each fulfilling the same purpose (defining a term).

benefits from parameterization

The data has enough structure that it is best modeled using clusters of key-value pairs. We explore this concept momentarily in [{XREF_data-params-tokens}].

All information that qualifies as small data should be organized in YAML or CSV files. It will either be prebuilt into includable content (snippets) or flattened into simple variables accessible by documentarians in AsciiDoc files.

Source Prebuilding: Liquid

The third markup language in the AJYL toolchain is Liquid. LDCMF documentarians typically do not write much Liquid, but it can be helpful to understand what it is for, how it is employed, and where the templates are stored. In the LiquiDoc toolchain, Liquid templates serve two purposes: **source prebuilding** via LiquiDoc and **site theming** via Jekyll.

prebuilding

Using Liquid templates to press small data into new source files (usually YAML or AsciiDoc) in preparation for further parsing and rendering.

theming

Code used for styling and shaping output artifacts. In LiquiDoc CMF and AsciiDoc/Jekyll projects generally, *theming* code is kept separate from *source matter* (content and data), mainly so content can be created agnostic to the "look and feel" it will take in various possible output formats.

Liquid templates used for prebuilding are stored in `templates/liquid/, and by LDCMF _convention` their file extension is the most explicit form of the target file extension. Here you will find `company-info.asciidoc`, which converts YAML data into AsciiDoc-formatted files with the `.adoc` extension. Likeqise, `xref-attributes.yaml` generates a new YAML file with a `.yml` extension.

Liquid templates can also be found in the `theme/<theme-name>/_layouts/` and `theme/<theme-name>/_includes/` directories. These will reflect `.html` extensions, as they are used to generate page templates and partials for HTML rendering.

In the interest of keeping content and presentation source separate, content source files (AsciiDoc) should never contain Liquid markup. Prebuilding maintains the integrity of the content source markup format (AsciiDoc only) and properly segregates page layout and formatting elements from

content and data.

Writing & Editing Content with AsciiDoc

AsciiDoc and Asciidoctor

AsciiDoc Basics

Syntax

Literal Expression in LiquiDoc Jargon

Text that appears inside literals is typically not processed by Asciidoctor during rendering operations. We have to signal the desire to parse variables inside literals.

Asciidoctor will not parse any non-AsciiDoc code. Even syntax highlighting of code examples is handled outside AsciiDoc, other than an indication of which language to highlight.

To indicate that Asciidoctor should parse attributes inside a code listing, flag it with `subs="+attributes"`.

```
[source,asciidoc,subs="+attributes"]
----
We can express a {version-number} inside our code listings as long as we've indicated
we wish to do so.
----
This matters not least because we will need to escape any existing strings in our code
samples that happen to contain braces separated by alphanumeric characters, hyphens,
or underscores (regex roughly: `[a-zA-Z0-9_\-]{4,50}`).
```

For one-line examples with no syntax highlighting, merely indent your code by one character.

```
 some shell command
```

```
[source,shell]
Line one of many
Line two of many

This line will not be expressed as part of the listing block.
```

```
[source,shell]
----
#!/bin/sh

some script down here
----

This line will not be expressed as part of the listing block.
```

Writing & Editing Small Data with YAML

Working with AsciiDoc Attributes & Liquid Variables

Locating & Managing Source Files

Understanding the architecture of any file- or content-management scheme takes time. Clarity and accessibility are aided by the strict use of conventions and standards shared *team-wide*.

So that you can **find content and data** when you need it, and so you can **determine where to save new content** as you create it, we offer LiquiDoc CMF's content and file management conventions. These are the rules governing how and where to store content and data of various types.

Discerning Content vs Data

Managing Content

Managing Data

Cross-referencing Topics with Built AsciiDoc Attributes

Cross referencing is an important element of technical documentation, but handling it has never been trivial. Cross references (“xrefs”) work differently for different document types, which adds some complexity

This section *extends* standard AsciiDoc usage with instructions set up to work in a properly configured LiquiDoc CMF environment. These practices may not apply in all AsciiDoc environments. For more, see the Asciidoctor User Manual on cross references.

Tokenized Xrefs

LiquiDoc CMF makes linking between units of content called “topics” fairly straightforward by generating attributes (variables) that can be used to write AsciiDoc’s complex cross-reference markup fairly simple.

For example, the markup `<<some-topic-slug#,That Topic’s Title Text>>` will be generated from `<<{XREF_some-topic-slug}>>`, because `xref`-prefixed variables have been generated for each topic slug.

While Asciidoctor rendering does automatically insert header text for other sections *within the document*, this does not apply to multi-page website artifacts. Asciidoctor-enhanced Jekyll builds a multi-document docset, whereas Asciidoctor itself generates single-document docsets in one-page HTML or multi-page PDF.

Explicit Xrefs

If you want to have *customized hyperlinked text* for a cross reference, use explicit syntax. You may have noted standard Jekyll xrefs require a hash mark to denote the previous string was a page slug and not an internal (same-page) reference, which now must be addressed in our source.

Example explicit cross references

```
Let's link to <<another-section-on-this-page>> and then to <<another-topic{sfx},custom
text hyperlinked to another topic>>.
```

These inter-page xrefs must be maintained manually to ensure explicit titles/headers are updated consistently whenever changes are made.

Because of the noted differences between Asciidoctor-enhanced Jekyll and conventional Asciidoctor document handling, xrefs to subsections inside topics does not work. A system of cataloging topic subsections (and other anchor points) will be required in order for any xref replacement system to work.

Submitting Data & Content for Review

Reviewing Merge Requests

Troubleshooting LiquiDoc Builds

Appendices

Appendix A: Jargon Guide

This is the full list of specialized terms used in this product documentation. They are also generated as JSON at `/data/terms.json` so we can highlight them in the text when we get to it. This is just to show the power of storing data in flat files reusable throughout product docs.

AJYL docstack

A combination of technologies (Asciidoctor, Jekyll, YAML, and Liquid) ideal for managing highly complex single-sourced technical documentation needs. (Resource)

artifact

A digital package (file or archive) representing a discrete component of a product. Here we use *artifact* to describe a discrete instance of final content output, such as a single HTML or PDF file, or a Jekyll website or Deck.js slide presentation.

AsciiDoc

Dynamic, lightweight markup language for developing rich, complex documentation projects in flat files. (Resource)

Asciidoctor

Suite of open source tools used to process AsciiDoc markup into various rendered output formats. (Resource)

build

As a *noun*, the (usually automated) series of actions necessary to compile and package software or documentation artifacts. As a *verb*, the act of performing a build operation.

build config

Refers to the file that defines a LiquiDoc build routine. I.e., {config_path}.

code source

In LiquiDoc projects, *code* refers to markup other than data and content source. For instance, theming templates are code, as are config files.

content source

Material, mostly formatted in AsciiDoc, including pages, topics, and snippets, but also including image assets that pertain to the project's subject matter, such as illustrations and diagrams (as opposed to themeing assets).

data source

Structured information, usually in YAML format, used to define variables, which can replace tokens in Liquid templates and AsciiDoc content source.

DocOps

An engineering discipline that focuses on integrated tooling and workflows to create optimal documentation environments. Similar to and derived from DevOps. (Resource)

docset

A collection of technical documents sourced from the same codebase, covering generally the same subject through different editions, possibly in multiple versions of each in multiple formats. For instance, an Administrator Manual and a User Manual sourced in the same Git repository, with overlapping content, each generated in HTML and PDF.

DRY

Acronym for “don’t repeat yourself”; refers to techniques for single-sourcing content and data so no information, illustration, explanation, etc is repeated in the source (threatening divergence).

Liquid

Open source templating markup language maintained by Shopify (Resource)

prebuilding

Using Liquid templates to press small data into new source files (usually YAML or AsciiDoc) in preparation for further parsing and rendering.

prime

As in *prime edition* or *prime repository*, references the *canonical* edition or source of a particular docset that has been forked. The *prime repo* of the LiquiDoc/LDCMF User Guides project can be forked and adapted to suit *your project’s* needs.

slug

A unique identifier made only of lowercase alphanumeric characters, as well as - (hyphen) and _ (underscore) symbols.

source matter

Either or both of *content* and *data*; any plaintext source files or database records used *in* the substance of generated output, therefore not including assets such as layout images or theming code.

theming

Code used for styling and shaping output artifacts. In LiquiDoc CMF and AsciiDoc/Jekyll projects generally, *theming* code is kept separate from *source matter* (content and data), mainly so content can be created agnostic to the “look and feel” it will take in various possible output formats.

YAML

A slightly dynamic, semi-structured data format for key-value pairs and nested objects (Resource)

Appendix B: How This Documentation is Built

LiquiDoc's own documentation is a fairly complex implementation of LiquiDoc and the LiquiDoc Content Management Framework (LDCMF). It takes advantage of most of LiquiDoc's capabilities and is the defining project for LDCMF.

The build is defined in `_configs/build-docs.yml`, which is a self-documenting configuration. Managing LiquiDoc build configurations is programming, albeit using an extraordinarily orderly "DSL" (domain-specific language). If you are not a developer, LiquiDoc's self-documentation features may seem more intimidating than helpful. Nevertheless, spending a few moments on this page and reviewing the LiquiDoc Docs configuration file may be the best way to get a sense for the power and dexterity of LiquiDoc.

Order Out of Chaos

LiquiDoc enables single sourcing of content and data by enabling files to be written to an effemeral directory.

The LiquiDoc configuration file is a map that pulls disparate files together just so, with the end result being one or more rich-media documents. LiquiDoc "steps" through this configuration when you run a build, and each step and substep yields automatic or custom messages. By default, these are printed to a file at `_build/pre/config-explainer.adoc`, or printed to screen with the `--explicit` command-line flag.

Go ahead and give it a try now:

```
bundle exec liquidoc -c _configs/build-docs.yml --explicit
```

This output is formatted as AsciiDoc ordered and unordered lists. You may find it helpful in understanding what the config file (`_configs/build-docs.yml`) is up to.

Appendix C: NOTICE of Packaged Dependencies

The following open source packages are fully or partially included with LiquiDoc.

Package	**Jekyll Documentation Theme**
License	MIT
Author	Tom Johnson
Website	https://github.com/tomjoht/documentation-theme-jekyll

Package	**M+ OUTLINE FONTS (M+ TESTFLIGHT 058)**
License	unlimited
Author	M+ Fonts Project
Website	http://mplus-fonts.osdn.jp/about-en.html

Package	**Noto Fonts**
License	SIL OFL
Author	Google i18n
Website	https://www.google.com/get/noto/

Package	**Font Awesome**
License	SIL OFL 1.1
Author	Fonticons, Inc
Website	https://fontawesome.com/

Package	**"Coding Style Guide"**
License	MIT
Author	Dan Allen, Paul Rayner, and the Asciidoctor Project
Website	https://github.com/asciidoctor/jekyll-asciidoc/blob/master/coding-style-guide.adoc