LiquiDoc Administrator Manual

toc::[]

Table of Contents

Getting Comfortable with LiquiDoc CMF . . . 3
- LDCMF: Not Your Granddad's Content Platform . . . 3
- LDCMF is AJYL . . . 4
- What Do I Need to Learn? . . . 4
- LiquiDoc and LDCMF Overview . . . 5
- GitHub . . . 6

Getting Started with the Documentation Codebase . . . 7
- Your Role . . . 7
- Installing Dependencies . . . 7
- Quickstart . . . 8

Establishing a Local Toolset . . . 12

Running LiquiDoc on Windows . . . 13

Understanding Your Role as Administrator . . . 14
- The Roles . . . 14
- The (Administrator) Role . . . 14

Getting to Know This LDCMF Environment . . . 15
- A Quick Review . . . 15
- Repo Tour . . . 15

Markup Basics . . . 19
- Content: AsciiDoc Basics . . . 19
- Small Data: YAML Basics . . . 19
- Source Prebuilding: Liquid . . . 21

Working with AsciiDoc Attributes & Liquid Variables . . . 22

Getting Comfortable with LiquiDoc Build Configuration . . . 23

Initializing a New LiquiDoc CMF Environment . . . 24

Handling Small Data . . . 25
- Data Sourcing . . . 25
- Parameterization . . . 25

Prebuilding Source with Liquid . . . 28
- Liquid Templates in Jekyll vs LiquiDoc . . . 28
- What Prebuilding is Good For . . . 28

Templating and Manipulating with Liquid . . . 30

Prebuilding AsciiDoc Source Files . . . 31
- When to Prebuild AsciiDoc Source . . . 31
- How to Prebuild AsciiDoc Source . . . 31

Prebuilding YAML Source Files . . . 35
- String Generation . . . 35

Migrating Assets During a Docs Build . . . 39

Converting Small-Data Files to Other Small-Data Formats . . . 40
Rendering Simple HTML5 Output . . . 41
Rendering PDF Output . . . 42
Generating a Static Website . . . 43
Theming Jekyll with Liquid . . . 44
Theming Jekyll with CSS . . . 45
Extending LiquiDoc's Capabilities Without Modifying Its Source . . . 46
Running LiquiDoc Using an Alternate Gem . . . 47
Install an older gem version . . . 47
Install a local modified gem . . . 47
Install a remote modified gem . . . 48
Troubleshooting LiquiDoc Builds . . . 49
Appendices . . . 50
Appendix A: Jargon Guide . . . 51
Appendix B: How This Documentation is Built . . . 53
Order Out of Chaos . . . 53
Appendix C: NOTICE of Packaged Dependencies . . . 54

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

Welcome to the LiquiDoc Administrator Manual!

Here you will learn to configure and develop the documentation environment for LiquiDoc and LiquiDoc CMF as an admin.

This guide exists to help you establish and maintain a complicated documentation project that essentially *treats docs source like product code*. You will learn to configure a complex docs project in a sensible format using LiquiDoc Content Management Framework (LDCMF). LDCMF combines the LiquiDoc open-source build utility with a set of standards and convention for organizing the codebase and proceding.

Hack These Docs!

As the LiquiDoc administrator, you are *strongly encouraged* to edit the source for your own implementation of this very set of documents, assuring that they convey information applicable to *your product environment*.

HEAVY META AHEAD

The following material is highly abstract. It relates to conceptualizing how the LiquiDoc/LDCMF User Guides repository can be adapted to suit your product and documentation environments. **This topic should be excluded** from or heavily edited for any instance of LiquiDoc Docs Project that is modified to cover a different product.

The original edition of this docset describes and instructs the management of LiquiDoc itself; as the administrator of a distinct product's documentation, you can reorient this guide to suit your needs in *two key ways*.

Customize for your docs

Content in these docs oriented toward "documentarians" instructs the use of LDCMF, the platform your docs team now uses. Make a few adjustments to suit your instance, and — *voila!* — and your own internal documentation instructions are bootstrapped.

Customize for your dev team

The developer-oriented documents in this guide, which instruct how to develop *LiquiDoc and LDCMF itself*, can be replaced with style guides, workflow documents, and other resources oriented to *your team's actual environment*.

Customize for your product

What you will actually find in this repository is a docset that instructs

You are in fact encouraged to modify a fork/clone of this repository such that it fits your own product. See the complete guide to adapting these docs to your project.

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 administrator
- get familiar with this LDCMF environment
- grasp the differences between content and data formats
- employ token substitution with Liquid variables and AsciiDoc attributes
- get comfortable with LiquiDoc build configuration
- initialize a new LiquiDoc CMF environment
- manage and use small data with parameters and tokens
- understand source prebuilding techniques
- understand how and where Liquid is used with LiquiDoc
- understand AsciiDoc prebuilding use cases, strategy, and techniques
- understand YAML prebuilding use cases, strategy, and techniques
- copy assets during a docs build
- convert small-data files to other formats
- render simple HTML output
- render PDF output
- generate a static website
- theme your Jekyll-generated guides portal using Liquid templates
- theme your Jekyll-generated guides portal using CSS
- handle product repositories as submodules
- classify and analyze the forms of documentation divergence
- implement appropriate solutions to manage divergence in the source
- adapt these very docs to cover my own environment
- extend LiquiDoc's capabilities without modifying its source
- execute LiquiDoc using an alternate gem
- follow clear steps to get unstuck when problems arise

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 (Administrator). See Understanding Your Role as Administrator to be sure.

As a LiquiDoc instance administrator, you'll need to intimately understand the relationship between LiquiDoc and LDCMF. The use of Liquid templating will likely be a significant part of your job, with the rest of your technical work being configuration and content management. You will also need to be familiar with the ***documentarian*** role, as covered in the LDCMF Documentarian Guide.

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.

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 an LDCMF *admin* for your own project, you will have to at least initialize an LDCMF environment before building any docs or even sharing the environment with your team. This section will get you started, and the rest of this Manual will help you customize the environment to suit your product demands and your team's workflow.

Once we make sure you have the few prerequisites in place, you can build these docs.

For more on your role as Administrator, 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". Gems can be executed via command-line interface or via LiquiDoc's Ruby API, still under development. This means you can include LiquiDoc, Asciidoctor, or Jekyll into your own Ruby applications. The CLI also makes LiquiDoc available to any build or deployment utility that can work the command line.

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.

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 Administrator

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*.

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 Guides. You are currently reading the Administrator Manual.

The (Administrator) Role

As an administrator of the LiquiDoc/LDCMF User Guides codebase, your responsibilities are many. Because the infrastructure is complex and every documentarian will have opinions, administrators must referee style, formatting, placement, and other conventions.

Administrators are also responsible for maintaining the integrity of the docs build.

Admins are also expected to create and maintain Liquid templates for prebuilding. This necessitates familiarity with AsciiDoc and YAML, the latter of which you will use for many other reasons, as well. You may be able to get away with not really knowing AsciiDoc, as long as documentarians provide you with the proper AsciiDoc format they need from prebuilding operations.

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.

_configs/

This folder is where the brains go. The `build-docs.yml` config file belongs here, as does anything that is more about programming the build procedure than about informing the content. The `asciidoctor.yml` file is for non-content AsciiDoc attributes that pertain to the structure or process of rendering with Asciidoctor. This is also the home of various Jekyll configuration files, usually one for each guide and one for each guide type (e.g., `attributes-portal.yml` and `attributes-manual.yml`).

_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.

As an administrator, you will need to mix all three markup languages. Definitely complete this {topic_unit}, but see Prebuilding Source with Liquid.

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.

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 Handling Small Data.

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. 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.

Working with AsciiDoc Attributes & Liquid Variables

Getting Comfortable with LiquiDoc Build Configuration

LiquiDoc configuration files are formatted in YAML. They represent a sequential procedure, with each item in a YAML array representing a step or substep of the build.

Let’s look at a very simple LD configuration.

Initializing a New LiquiDoc CMF Environment

Handling Small Data

Data management is one of the docs administrator's key responsibilities. In the LDCMF context, data management means ensuring that all data is properly organized and readily accessible. This includes not just source file management but also built-source generation, or prebuilding.

The documentarian need only know what variables are available and where they came from, the admin must concern oneself with up-front choices about data structure and placement that will ensure the widest usability. Sometimes this means manipulating raw source data into new forms of built data, but always this means making sure information gets stored in the right places, and that data objects or records are always only as complex as they need to be, but never too simple to be useful.

Data Sourcing

TODO: This section will describe what kinds of data goes where.

Parameterization

Data arranged in key-value pairs is considered *parameterized*. Review these three examples and their descriptions first, then we'll discuss the differences and applications for all three.

Example A — Structured content in AsciiDoc

```
=== Definitions

AsciiDoc:: A wicked cool, dynamic lightweight markup language
YAML:: A lightweight markup for data
```

Example B — Minimally parameterized data in YAML

```
definitions:
  AsciiDoc: A wicked cool, dynamic lightweight markup for structured writing
  YAML: A lightweight markup for data
```

Example C — Fully parameterized data in YAML

```
definitions:
  - term: AsciiDoc
    definition: A wicked cool, dynamic lightweight markup for structured writing
  - term: YAML
    definition: A lightweight markup for data
```

The first example (Example A — Structured content in AsciiDoc) is fairly useful. It tells the Asciidoctor rendering engine that the content should be rendered as a definition list, which can then be incorporated into PDF and CSS styling of the resulting document. Without us configuring anything else, this source would render a nice little definition list — something like:

Example 1. Minimally styled rendering

Definitions

AsciiDoc
: A wicked cool, dynamic lightweight markup language

YAML
: A lightweight markup for data

Unfortunately, each item can only be used in place, at least without much more manual work to enable it to appear in other AsciiDoc files — even then, it cannot be reshaped (such as into table cells instead of a definition list). Using AsciiDoc to define small data within the file adds lots of effort as well as complication to AsciiDoc writing, without adding advantages for reuse outside this docset. With product info in semi-structured formats like YAML, that data becomes available externally, such as for product code or separate docs.

The second example (Example B — Minimally parameterized data in YAML) is a little more useful, in that it could be accessed by external resources as well as our AsciiDoc files. It matches our earlier criteria for Simple variables in YAML, in that they are just a key and a value, with no additional structure and fairly simple strings as values.

In fact, this information is most usefully stored as an *array* of parameters, as we see in the third example (Example C — Fully parameterized data in YAML). This means we treat each term as a value associated with a key called `term`, and we still treat the definition as a value, now for a key called `definition`.

An array divides its immediate contents into a list of items, but each item can contain many *parameters*, or *key-value pairs*. This is serialized data at its most elegant. Data formatted like this is much easier to work with in templates, as we will not need to perform any special transformation on the elements in order to rearrange them.

This structure gives administrators significant programmatic potential, but it also gives documentarians an elegant way to work with serialized data independent from the pages it will eventually appear in. All this without the overhead of databases designed for much larger and more-complex forms of data, and without their clunky interfaces, to boot.

This format also enables us to add additonal parameters to each or any element down the line without having to restructure the whole collection.

Fully parameterized data, now with categories

```
- term: AsciiDoc
  definition: A wicked cool, dynamic lightweight markup for structured writing
  categories: markup, writing
- term: YAML
  definition: A lightweight markup for data
  categories: markup, data
```

We revisit the way small data is handled in <<(XREF_source-prebuild-basics}>> and Theming Jekyll with Liquid.

Prebuilding Source with Liquid

A key strategy of LiquiDoc CMF-based documentation platforms is prebuilding content from semi-structured data sources. That is, we form some source content out of prior source content. For instance, we can ingest a JSON-formatted file and turn its structured contents into variables that are used throughout our content and even our site navigation and layout.

This approach allows a stable, linear construction of multiple, “parallel” docsets from roughly the same source, all in the same codebase. The prebuilt source files used for final Asciidoctor- and Jekyll-driven rendering operations are themselves artifacts of a prepared build. They can be examined during troubleshooting like a multi-dimensional stacktrace just by navigating the `_build/`. This prebuilt content can also be used as the direct source for conventional Asciidoctor and other static-site build commands as part of an alternate toolchain.

Liquid Templates in Jekyll vs LiquiDoc

As will be reiterated frequently, we use files written in Liquid templating format to shape our output in two key ways. First, we use it for **prebuilding**, usually to generate both AsciiDoc and YAML files out of earlier source files. Second, we use it to generate elements of the **layout and metadata** of the Jekyll websites we generate during the later render stage.

Prebuilding AsciiDoc and YAML from the `_templates/liquid/` directory is coordinated by LiquiDoc during `parse` actions. Prebuilding of (mostly HTML) files in the `theme/docutheme/_layouts/` directory is performed natively by Jekyll during the Jekyll `render` stage of the LiquiDoc build.

We do *not* use Liquid markup in AsciiDoc files for rendering during Jekyll builds, even though the jekyll-asciidoc plugin does enable parsing of Liquid inside AsciiDoc. Prebuilding via LiquiDoc is the preferred method of structuring AsciiDoc source using Liquid templates.

What Prebuilding is Good For

This strategy is usually only advantageous under two conditions:

1. When maintaining **multiple output artifacts** that vary in their content, not just their format, prebuilding allows you to establish segregated sets of variables that apply only to each given artifact.
2. When you want to share product information between product source code and doc source, **single-sourcing** this information in semi-structured data files allows the team to maintain one single source of truth.

Content variables (AsciiDoc *attributes* you call out with `{attribute_name}` inside your content) are most useful under these circumstances.

The third use case for content variables in AsciiDoc is typically just to store repeatedly used data all in one spot — that is, single sourcing across docs. This would be either in a small-data file as described above, or as lots of attributes set at the top of the AsciiDoc index file (**inline definition**).

These examples do *not* constitute AsciiDoc prebuilding, as we are passing variables directly into AsciiDoc during the render operation.

Tooling clarity

When you perform `asciidoctor` operations via LiquiDoc, you can load attributes from external YAML files (a function of LiquiDoc) *and* use inline definition (supported by Asciidoctor). However, storing variable data in YAML files is preferred because it centralizes important data instead of sprinkling it at the top of individual topic files. This external-datasourcing feature is not natively supported by Asciidoctor's own tooling.

Templating and Manipulating with Liquid

LiquiDoc CMF ***administrators*** will likely spend considerable time wading through Liquid template files, either to *generate* (1) layout, structural, and metadata elements of the *website* or (2) iterative elements of the *content* (prebuilding). In each case, templating serves the purpose of mixing small data and *tokenized* markup, substituting *parameterized* data for embedded tokens.

Liquid is a fairly light, tag-based markup format that enables conditional flows, iteration/looping, and powerful variable substitution.

Example — Conditionally looping variables in Liquid

```
{% for p in pages %}
{% if p.categories contains "tutorials" %}
<li><a href="{{ p.url }}">{{ p.title }}</a></li>
{% endif %}
{% endfor %}
```

We'll delve deeper into Liquid later on (see Theming Jekyll with Liquid), but it is probably already becoming clear that we need to get all these different kinds of variables under control.

Prebuilding AsciiDoc Source Files

When to Prebuild AsciiDoc Source

How to Prebuild AsciiDoc Source

Variable Parsing

Prebuilding involves templates, used to “press” data into another textual output format. This output can be any “flat” format, meaning readily savable in a text file as opposed to a complicated binary format. Everything from Markdown to XML is a flat format, and HTML is by far the most popular target format for templating engines.

In our case, the output format is AsciiDoc, a format itself associated with **early source** (like Markdown, popularly rendered into HTML for final rendering by a browser) and not **late source** (like HTML, directly rendered by the browser with no need for intermediary processing).

In your LiquiDoc config file, prebuilding looks something like this:

```
- source: data/mydata.yml
  builds:
    - template: _templates/liquid/mytemplate.asciidoc
      output: _build/includes/_built_my-include.adoc
```

Since this project builds separate but heavily overlapping sites from a singular codebase, variables are extremely useful. AsciiDoc and Liquid are both *tokenized* markup formats. They both provide markup for variable substitution, whereby a token is replaced by its expressed value according to small data passed in during parsing.

In AsciiDoc files, such variables are called *attributes*, and they are simply wrapped in single curly braces:

```
The default is set to `{sysmem_default}`.
```

In Liquid templates, variables are wrapped in double curly braces. Here is an example from the Liquid template used to generate header info for topics. This template provides the model for generating AsciiDoc-formatted output from the small data in `data/schema.yml`.

Sample from data/schema.yml

```
topics:
  - title: Yocto in a nutshell
    slug: yocto_c_nutshell
    portals: all
  - title: Yocto application development
    slug: yocto_r_application-development
    portals: all
```

topic-page-meta.asciidoc Liquid template

```
{% for topic in topics %}
// tag::{{ topic.slug }}[]
= {{ topic.title }}
:page-title: {{ topic.title }}
:page-permalink: {product_slug}/{{ topic.slug }}
// end::{{ topic.slug }}[]
{% endfor %}
```

The first line initiates a loop procedure to iterate through the given data effectively, as we'll explore shortly.

The 1-space padding around the token string in Liquid variables (ex: `{{ topic.title }}`) is conventional but not required.

This is pretty much as powerful as it gets in AsciiDoc, but Liquid offers some additional capability. Because Liquid can keep track of nested variable structures, its variable references can have multiple tiers.

These can be represented with dots denoting the full data hierarchy path. Let's use data arranged in a nested fashion, like so.

Example dummy-data.yml

```
level_one:
  level_two:
    - name: item-1
      number: 1
      tags:
        - tag1
        - tag2
    - name: item-2
      number: 2
      tags:
        - tag2
```

Here are some data expressions we can derive from this sample data in Liquid:

```
* {{ level_one.level_two[1].tags[0] }}

* {{ level_one.level_two[0].tags[1] }}
```

If you are familiar with arrays, take a moment to see if you can figure out what those two bullet points should resolve to. The answer is immediately below.

For anyone who is not already handy with arrays, let's look at the resolution first, and see if you can figure out how arrays work based on this resulting output.

```
* tag2

* tag2
```

In most software languages (including Ruby and Liquid), array items are numbered starting with index slot `0`. Brackets are used to indicate array index slots. Since we have two nested arrays, we read the block hierarchy (`level_one.level_two`, where `level_two` is the first array), then we list which index slot we want to express.

Let's take the first variable name: `level_one.level_two[1].tags[0]`. By indicating we want the item at index slot `level_two[1]`, we ask for the *second* item in the array named `level_two`. This happens to be the item with `name: item-2`. Next, `tags[0]` calls for the *first* item in that array's `tags:` block, which happens to be the only item, `tag2`.

The second variable, named `level_one.level_two[0].tags[1]`, gets the same result by asking for the *first* item in the first array (`item-1`) and the *second* item in the second array (`tag2`).

Any number of blocks can be called this way, enabling deeply nested YAML structures.

Iterating Through Data

The previous example from `data/schema.yml` used a Liquid `for` loop to iterate through serialized data. We use this feature to generate serialized output from multiple items in a list or array. It can generate an "unordered" list of bullet points or menu items just as surely as it can output a series of table rows.

This looping feature is only available in Liquid templates, not AsciiDoc templates, at this time. This is why AsciiDoc prebuilding happens outside and prior to the stage or stages during which we render with AsciiDoc into final output.

Let's try a looping example, this time on a chunk of familiar sample data.

Example dummy-data.yml

```
level_one:
  level_two:
    - name: item-1
      number: 1
      tags:
        - tag1
        - tag2
    - name: item-2
      number: 2
      tags:
        - tag2
```

Here is a way this can be expressed in Liquid:

```
{% for item in level_one.level_two %}
* {{ item.name }} ({{ item.tags[0] }})
{% endfor %}
```

This Liquid generates the following output:

```
* item-1 (tag1)

* item-2 (tag2)
```

Where we name our looping index variable `item`, we could be naming it `i` or `itm` or `idx` or `mrhooper` — we're just designating a name so we can reference its member variables (such as `name` and `tags`). This code will iterate through the two items in `level_one.level_two`. The variable `name` is a string in each instance, so the string is expressed. The variable `tags` is an array, and we're looking for just the first item in that array by calling for `item[0]`. This time we get divergent results by asking for the exact same index slot in each `tags` array, since each array has a different value in that slot.

Prebuilding YAML Source Files

We use YAML prebuilding to create new data sources from existing ones. Original sources can be anything, but we aim for YAML output so we can re-use the data for other operations.

The `_templates/liquid/nav-sidebar-prtl.yaml` file is an example of prebuilding data files from other data files. Look inside `_build/data/built/nav/` — you'll find files built from `data/schema.yml` using the above `.yaml` Liquid template during prebuilding: `devdocproadmin.yml`. These files are arranged such that additional (Jekyll) templating can convert them into HTML for its corresponding portal's nav menu. See [{XREF_theming-jekyll-liquid] for more on post-processing with Liquid during a Jekyll render phase.

This is YAML prebuilding: the generation of YAML source files from other datasources, pressed into new/dependent forms using Liquid templates.

In addition to the prebuilt sidebar data files, one of the more interesting use cases for YAML prebuilding is reflected in the `_build/data/built_x-platform.yml` file. This file allows us to include any topic's counterparts across platforms. Since we don't have a relational database to query, we reinterpret our own data from `data/schema.yml` so that we can insert links to a representation of the same topic in another portal (documenting another platform). The template `_templates/liquid/x-platform-rels.yaml` contains the fairly complex programmatic logic that performs that interpretation and orchestrates the dependent `build_x-platform.yml` output.

String Generation

Another key implementation of YAML prebuilding is string generation. The prefered method for handling this is to generate string variables/attributes for use in AsciiDoc or Jekyl code during rendering, which we'll explore in the next two subsections.

For instance, using YAML, we can create portal-specific *compound variables*, concatenated from previouly sourced data during prebuilding. Also, some variables need to work differently in different portals. We'll call these Split-expression Variables.

Both types of strings are constructed in a Liquid template called `data/string_vars.yaml`, which generates `_build/data/built/strings.yml`. The parameters in `string.yml` are ingested by Asciidoctor and made available in AsciiDoc templates as `{variable-name}`. They are not available, however, in Jekyll templates, as they cannot yet be ingested per-portal. Remember, Jekyll variables are for navigation and contextualization, not for product details.

Compound (Concatenated) Variables

We store core product info in the `data/products.yml` file, but dependent data parameters can be dynamically generated from the original data and through concatenation. YAML's dynamic capabilities are poor, so we add the power of Liquid templates to generate whole new values to assign to new keys.

Take this URL, for example: `ftp://ftp1.example.com/support/digiembeddedyocto/2.4/r3/images/ccimx6ulsbc-installer.zip`. This is the toolchain installer for the 6UL Pro. In AsciiDoc, we use the

attribute token `{toolchain-url}` to express this variable. That variable is constructed from three other variables expressed as Liquid tokens, along with some hard-coded strings to tie it all together.

From _templates/liquid/string_vars.yaml — the default/Pro toolchain URL

```
  toolchain-url: ftp://ftp1.example.com/support/digiembeddedyocto/{{prod.dey-
version}}/{{prod.dey-release}}/images/{{prod.prod-codename-pro}}-installer.zip
```

Inside `_templates/liquid/string_vars.yaml`, we loop through each portal's setting from `data/products.yml` using a scope called `prod.`, which you can use to express a portal's settings in YAML format. Here the double-braced tokens set each platform's DEY version, release, and platform indicator strings.

The parameter keys can also be made dynamically, if the need ever arises. Simply add Liquid tokens into the key portion of the `string_vars.yaml` file.

Split-expression Variables

Some ConnectCore platforms have both a Pro and Express board, each with different features. For this reason, there is sometimes structural divergence between how we would reference a 1-board portal vs a 2-board portal, meaning the same question (e.g., "What is the platform indicator code?") might have more than one answer for a given platform, which we would want to present together.

One way we handle this is by **creating additional variables** so that we can express more than one value in the appropriate environment, which we will perform manually using a conditional.

Remember our pro/default `{toolchain-url}` attribute? For certain platforms, we also need this URL for the Express board's toolchain. In that case, we'll use `{toolchain-exp-url}`, which we'll set using the alternate platform indicator: `{{prod.prod-codename-exp}}` to form a unique value.

From _templates/liquid/string_vars.yaml — the Express toolchain URL

```
{%- if prod.prod-exp %}
  toolchain-exp-url: ftp://ftp1.example.com/support/digiembeddedyocto/{{prod.dey-
version}}/{{prod.dey-release}}/images/{{prod.prod-codename-exp}}-installer.zip
{% endif -%}
```

This is a conditionalized compound variable. The *condition* is whether an Express option exists on the platform. Inside `string_vars.yaml`, we see the Liquid conditional `{%- if prod.prod-exp %}`. The parameter `prod-exp` is only set (and thus only returns `true` in Liquid) for certain portals (guide-2 and portal-3), so this section will not be generated for the guide-1 portal. The next section of `string_vars.yaml`, by contrast, denoted by the `{%- unless prod.prod-exp %}`, is a very explicit way of doing the opposite: establishing settings *only* for the guide-1 portal.

When it comes time to express this second attribute in an AsciiDoc file, we need to be careful, as this attribute does not even exist on all platforms. In AsciiDoc, we use the macro `ifdef::prod-exp[]` to test for the existince of an Express option.

Sample AsciiDoc markup for presenting a conditionalized variable

```
* link:{toolchain-exp}[{prod-name-pro}]
```

The first line builds an link based on an attribute that exists in all portals. The second line establishes a condition. If this condition is *truthy* (the attribute is defined), we'll display the conditionalized content, in this case the link for the Express toolchain Don't forge to close your conditionals with `endif::[]`.

Variables will only test as *defined* when they've been set for the portal being rendered. This is true for AsciiDoc (`ifdef::variable-name[]`) as well as for Liquid (`{% if variable-name %}`). So in cases like `prod-exp`, we want to leave the variables undefined (in `data.yml` and prebuilt output `built_strings.yml`) for the portals where they do not apply.

Another approach is to construct a variable that substitutes different combinations of variables depending on the portal. The AsciiDoc attribute `{boards-and}` is such a variable. In the portal-3 portal, it resolves to `ConnectCore 8X SBC Express and SBC Pro boards`, while in the guide-1 portal it resolves to `ConnectCore 6 SBC board`.

This way we can set the value of `boards-and` two different ways for the different platform board arrangements. Here is where the compound part comes in. We want `boards-and` to express both boards for the two-board platforms, so we establish it as:

From `string_vars.yaml`

```
{% if prod.prod-exp %}
  ...
  boards-and: {{prod.prod-name-som}} {{prod.prod-name-exp}} and {{prod.prod-name-pro}}
boards
  ...
{% endif %}
```

This would not make sense for the guide-1 portal, however, since the CC6 SBC offering only has one board. So we place he code for generating that parameter in the *unless* section.

From `string_vars.yaml`

```
{% if prod.prod-exp %}
  ...
  boards-and: {{prod.prod-name-som}} {{prod.prod-name-pro}} board
  ...
{% endif %}
```

Now we can say things like `On the {boards-and}, you'll find⋯`.

Let' examine another example. In the `products.yml` file, all three portals have a setting called `prod-filesystem`, though the value in guide-2 and portal-3 (`ubifs`) differs from the value in guide-1 (`vfat`).

What if we wanted to divide portals according to whether their platform uses vfat or ubifs, just the way these platforms are already divided by whether they have an Express option? Maybe we want to be able to conditionalize content around this divide.

When prebuilding the YAML our string data file, we conditionalize by platform.

```
{% if prod.prod-fileystem == "vfat" %}
  fs-vfat: true
{% elseif prod.prod-filesystem == "ubifs" %}
  fs-ubifs: true
{% endif %}
```

Now we can segregate content anywhere in our AsciiDoc files using the `ifdef` macro.

```
ifdef::fs-vfat[]
Content that applies only to vfat filesytems.
endif::[]
```

Migrating Assets During a Docs Build

Converting Small-Data Files to Other Small-Data Formats

Rendering Simple HTML5 Output

Rendering PDF Output

Generating a Static Website

Theming Jekyll with Liquid

Jekyll uses Liquid templates for page structure and for generating elements such as nav menus. Liquid templates enable data iteration (looping), conditional flows (including switches), variables with transformation (filtering).

Jekyll's particular (modified) implementation of Liquid's templating system is covered in the Jekyll docs. The official Liquid for Designers wiki is very good, but the official Liquid manual is also helpful.

Liquid templates in the `theme/<theme>/_layouts/` and `theme/<theme>/_includes` directories can draw on data from any YAML files in the `data/` directory. These variable names are structured as: `site.data.<datafile>.<varpath>.<varname>`, where <datafile> is the name of the file in the `data/` directory, where `<varpath>` is any further hierarchy inside which our intended variable is nested, and where `<varname>` is our final key.

Unlike with content/data prebuilding, Jekyll theme processing data must be sourced in YAML only. It can contain strings, numerals, arrays, and structs. The latter data types apply hierarchical nesting, as they imply subordinate data.

Also unlike with content/data prebuilding, layout and include templates all have a `.html` file extension but are called merely using their extensionless filename.

During LiquiDoc build procedures, the data directory used will be `_build/data/`. This will contain the migrated contents of the root `data/` dir as well as prebuilt YAML files, all centrally located for Jekyll ingest.

Theming Jekyll with CSS

Jekyll incorporates SASS compilation out of the box, but a LESS converter can be readily added. Our current theme is not Sassified or Lessified yet.

Extending LiquiDoc’s Capabilities Without Modifying Its Source

Running LiquiDoc Using an Alternate Gem

There are two fairly common cases in which you might wish to run a version of LiquiDoc other than the latest officially released edition. You may need a previously released version due to deprecated functionality or an unfixed bug in the latest release. Or maybe there is functionality you need that only exists in unreleased version — probably a version you've hacked yourself or one already pushed to GitHub but awaiting release.

Here we instruct all three use cases: previous release, local modification, and remote modification. Each method recommends editing the `Gemfile` in your root directory. Open it with your favorite code/text editor, and remember to run `bundle update` after re-pointing your `liquidoc` dependency.

For advanced gem-version designation tricks, see Bundler's Gemfile documentation.

Install an older gem version

If you need to invoke an older version of LiquiDoc, simply designate the required version and update your dependencies.

1. In your `Gemfile`, edit the `liquidoc` line by setting a required version of the gem.

```
'liquidoc', '0.6.0'
```

2. Save your modified Gemfile.
3. Run `bundle update` on the command line.

Install a local modified gem

If you have a modified clone of the `liquidoc-gem` repository on your local system, Bundler will build the gem at runtime as long as you have the `liquidoc` dependency properly designated.

1. In your `Gemfile`, edit the `liquidoc` line by adding a path to your local `liquidoc-gem` repo.

```
'liquidoc', path: '../liquidoc-gem'
```

The pass value can be absolute or relative to the `Gemfile` itself.

2. Save your modified Gemfile.
3. Run `bundle update` on the command line.

Subsequent `liquidoc` commands will use this alternate source.

Install a remote modified gem

If you need to run a pre-release version of LiquiDoc that is posted in a remote repo, such as in a branch in the prime repo that has been submitted but not

1. In your `Gemfile`, edit the `liquidoc` entry to designate a Git repo and a specific branch,

   ```
   'liquidoc', :git => "{github_git_uri}", :branch => "special-mode"
   ```

   Where `special-mode` is the name of an unmerged branch you want your gem built from.

   

   Instead of a branch, you can designate a specific revision hash with `:ref ⇒ "a3iq0k"`, where `a3iq0k` is an example partial hash, enabling you to build a gem from any past commit. As another option, specific Git tags can be designated with a notation such as `:tag ⇒ "v1.0.0-rc"`.

2. Save your modified Gemfile.
3. Run `bundle update` on the command line.

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