DITA Migration Guide

Contents

Introduction

One of the mantras in the Republic of DITA is that "DITA is both highly beneficial and highly disruptive."

Organizations considering a migration to OASIS DITA tend to focus on the former – the many technical benefits that DITA can bring to a group of content developers interested in reuse, consistency, style/content separation, and content management. The technical merits of DITA are reasonably well documented and exemplified, especially for enterprise-class organizations. Equipped with a few books and a decent DITA editor, teams can produce some technically compelling demo implementations of unstructured content converted to DITA, replete with reuse, metadata, topic authoring, multi-channel publishing, and so forth. Less visible to organizations considering DITA migration are its "highly disruptive" aspects – content upheaval, role redefinition, staff attrition, sacred cow burgers, and politics. The long-term costs of ignoring the non-technical, disruptive aspects of DITA migration when building a proposal for migration can be damaging if not fatal.

Approaching a DITA migration with eyes wide open is necessary. DITA is an XML technology and a loosely identifiable set of best practices. There are no guarantees that it maps neatly into your organization, your management priorities, or your team culture. Reading a stack of DITA books and attending a bevvy of DITA webinars can be informative, but they are no substitute for learning from the experiences of other who have succeeded and failed in their DITA migrations. People involved with failed DITA migrations can tell you – in hindsight – that all the signs were present at the beginning of the migration that it was doomed organizationally or politically. Experience.

This *DITA Migration Guide* provides a framework within which the OASIS DITA Adoption Technical Committee can capture the collective wisdom and experience of the DITA community regarding DITA migrations. Welcome to the conversation.

This project, sponsored by the OASIS DITA Adoption Technical Committee, seeks to provide you with a lot of ready-to-reuse information about the many benefits of migrating to DITA *and* (importantly) some of the trade-offs and risks that also need to be put on the table. The strongest migration proposal for DITA is one that is balanced, data-driven, and sensitive to the disruption that DITA can cause.

The Glass Half Full . . .

Most DITA migration proposals seek funding or approval from senior management on the basis of the technical benefits offered by DITA. These benefits can be quite attarctive to organizations seeking any of the following:

- *Industry-standard markup and metadata*: Most desktop authoring tools and all of the "complete solution" suites are based on proprietary file/content formats, proprietary authoring interfaces, proprietary processing, and (often) proprietary content management. If you have developed your content in one of these environments, you are quite probably locked into the whole package – content format, tools, everything. Building your content on an open industry standard is the key. Any authoring, processing, or content management tools that are compatible with the underlying standard should be interchangeable (in theory). When it comes to DITA-compliant authoring, processing, and content management tools, "proprietary" is not a dirty word. Being able to switch out tools without having to reimplement reformat the underlying content is a big deal. DITA being an open industry standard supported my open-source and commercial tools developers protects an organization's investment in the underlying content.
- *Community development and support*: DITA has been in the market as an OASIS standard for over ten years. The likelihood that your organization is facing unique or significantly different requirements is probably small. In the cumulative library of relevant case studies, whitepapers, conference presentations, webinars, and user group notes, you *will* find information that is directly relevant to your particular market and/or audience. With so many companies using DITA, there are abundant training, consulting, and community resources available to you as you begin your investigation.

  Tip: To identify other companies that have adopted DITA in your region or in your market, consider browsing the listing at *Companies Using DITA*.

- *DITA sells itself*: In addition to this migration proposal domain, there are numerous sample domains of DITA source content. Without creating any new content, you can download some tools and sample sources in order to demonstrate how DITA works, how it behaves in an authoring tool, how processors generate multiple output types, and how its modular architecture supports agile content reuse. Some percentage of the people who are most skeptical about DITA have rarely or never touched it or seen it live.
- *DITA is for people too*: Yes, DITA can be a playground for XML nerds, but it is also something that has been deployed in real organizations with real people for a long time. The impact that DITA can have on writers, editors, managers, and architects is reasonably well known and documented in many contexts. Having a feel for those human costs (good, bad, and indifferent) is critical to your developing a well-rounded proposal. DITA certainly has its critics and detractors. Some of their arguments are, quite honestly, valid and some are specious. You should assume that the people on your team and elsewhere in your organization have heard the stories and will be more open to your argument if you, in turn, are open about the people stuff – roles, careers, recruiting, mentoring, productivity, resistance, change management, and so on. One of the two most common mistakes in selling a DITA migration involves a lack of investment in selling the people affected by the migration. Not all of them need to agree with you, but they are all stakeholders and need to be treated as such.

  Tip: A good place to start your investigation about DITA roles would be the OASIS DITA Adoption whitepaper *Roles and Responsibilities of a DITA Adoption*.
- Content reuse
- Lower translation costs
- Long-term writer productivity gains – do more with the same or fewer people
- Content management and sharing
- Output consistency across product lines and development sites
- Integration with corporate publishing portals
- Integration with other engineering infrastructure

Although this sounds a bit daunting, there is some really good news for you to consider up front.

The Glass Half Empty . . .

If 100% of the organizations beginning a DITA migration finished successfully, there would be no need conferences, user groups, webinars, or consultants. Before you invest significant time in championing DITA in your organization, please take stock of some of the inhibitors that may be lurking in your organization.

- zero-cost migration
- zero-attrition migration
- zero-disruption migration
- bad content miracle cure
- $0 budget
- Home-grown tooling
- Home-grown processes
- Special snowflakes
- One person's concept of waste . . .
- Feathered nests
- Tail of the dog
- The Maginot line

Lots TBD here . . .

How to Reuse Contents in This Guide

DITA is all about reuse. A significant amount of material in this guide may be useful to you as source material in crafting a DITA migration proposal in your own organization. Please do so. Here are a few tips that may help you in reusing these DITA sources.

Key names available in this domain

The DITA map named `mapkeys_dita-migration.ditamap` contains a collection DITA key definitions that are used throughout this domain. Key names are *italicized*, key values are **boldfaced**.

```
<keydef keys="k_kw_migrate_org-name"><topicmeta>
  <keywords><keyword>[YOUR-ORG-NAME]</keyword></keywords>
  </topicmeta></keydef>
<keydef keys="k_kw_migrate_product-name-1"><topicmeta>
  <keywords><keyword>[YOUR-PRODUCT-NAME]</keyword></keywords>
  </topicmeta></keydef>
<keydef keys="k_kw_migrate_product-name-2"><topicmeta>
  <keywords><keyword>[YOUR-PRODUCT-NAME-2]</keyword></keywords>
  </topicmeta></keydef>
<keydef keys="k_kw_migrate_proposal-title"><topicmeta>
  <keywords><keyword>[YOUR-PROPOSAL-TITLE]</keyword></keywords>
  </topicmeta></keydef>
```

If you update the value of one of these keys in this map, all references to that key name in topics throughout this domain will auto-magically update the next time you open a topic or generate output.

Try it.

Key name	Referenced key value
`k_kw_migrate_org-name`	[YOUR-ORG-NAME]
`k_kw_migrate_product-name-1`	[YOUR-PRODUCT-NAME]
`k_kw_migrate_product-name-2`	[YOUR-PRODUCT-NAME-2]
`k_kw_migrate_proposal-title`	[YOUR-PROPOSAL-TITLE]

Basic Concepts in XML DITA

Before you and your team begin your journey, consider reviewing some of the basic concepts underlying XML and DITA. Discovering where you team can achieve consensus and where you agree to disagree will help during subsequent phases of a migration when you need to make tradeoffs in your goals, design, or schedule.

Tip:

You will *always* need to make tradeoffs and compromises.

About XML Grammars, Vocabularies, DTDs, and Processors

XML has its own professional vocabulary for talking about the stuff in formal language specifications. The audience for the OASIS DITA specification includes tools developers, consultants, and system integrators, not writers, editors, managers, or information architects. Language specifications require that level of precision, chances are that most people considering a DITA migration do not. This topic attempts to describe some of the basic concepts of the XML language and architecture in terms of something that we all know – natural language.

Grammar

Proficient speakers in any natural language such as English, Chinese, or Navaho know that sentences are constructed according to a set of rules. English sentences that do not conform to the subject-verb-object sequence are irregular

(Yoda-speak). Regardless of what words are used in the sentences, the sentence structures for any language need to follow a grammar in order to make sense.

Computer markup languages such as HTML, XML, wiki, Markdown, and so on rely on conventions for how they encode content. The set of encoding rules for HTML4, for example, do not require that every opening tag (element) have a closing tag (element).

```
<li>This is an item in an ordered (numbered) list in HTML4.
```

Similarly, most lightweight markup languages such as Markdown require only opening tags.

```
# This is an item in an ordered (numbered) list.
```

In more structured markup grammars such as XHTML, HTML5, or XML DITA, every opening tag must have a corresponding closing tag to be considered "well formed".

```
<li>This is an item in an ordered (numbered) list.</li>
```

When a web browser or markup parser reads an HTML, Markup, or XML document, it first evaluates whether the encoding conforms to the grammar that it expects. On this first pass, the parser is not evaluating elements or attributes or metadata, it is purely evaluating the markup conventions used in the document. If the document has markup that does not conform to the HTML, Markdown, or XML grammar rules, the document can be rejected by the parser and non-conforming or "not well formed".

Vocabularies

Although children need no more than a couple of years to master the grammar for their first language, they may spend many years learning the words (vocabulary) that comprise conversational and professional discourse.

Once you have learned a set of encoding rules (grammar) for a computer markup language, you need to wrapper or encode your content in a set of tags (elements) specific to that language. Lightweight markup vocabularies such as Markdown rely on a small set of easy-to-remember, easy-to-use tags: # (Heading-1), ## (Heading-2), * (bullet list item), and ` ` (code phrase). More sophisticated markup languages such as XML DocBook or DITA have hundreds of elements designed to capture complex structures (nested s) and semantic content (<msgph>). Different flavors of the same markup language usually share the same grammar but not the same vocabulary. What determines the complexity and breadth of a markup language is a combination of its out-of-the-box vocabulary and its capacity for extending that base vocabulary. We have, for example, dozens of variations of wiki and Markdown because their base vocabularies could not be extended – only replaced or modified. By contrast, the DITA specializations are all extensions of the base DITA vocabulary.

DTDs (Document Type Definitions)

Structured markup languages such as SGML and XML rely on parsers to compare each document that we author against a strict "definition" of the vocabularies allowed in a document. If the Document Type Definition (DTD) for DITA task topics specifies, for example, that the first element in that document be a <prolog> element, all my individual task topics must open with a <prolog> element. DTDs specify how the *langue* (superset) of abstract vocabulary elements must be combined and sequenced to produce a particular type of document (*parole*, implementation). The combination of defining vocabulary relationships in DTDs and validating (parsing) each instance of a document type (topic) against its DTD ensures that all the documents containing your content are consistent relative to the rules you establish in your DTDs. That's what we call *valid* in XML. Very strict or constrained DTD definitions produce more restrictive topics. If you are managing 100 topics, this level of consistency is a nice-to-have. With 100,000 topics in multiple languages, 100% validity and very high consistency are must-haves. Automated processing and publishing pipelines assume this level of validity and consistency.

Processors

Processors perform three tasks.

1. They read (parse) documents designated to be "built" or "rendered" in some output format.

2. They associate tags in the markup such as <b> with specific formatting in the target output (bold). The intermediate lookup table between markup tags and output formatting is implemented in XML as XSL files. Each type of output (PDF, HTML5, EPUB) requires a different set of XSL processor files.

   Note:

   Occasionally processors perform analysis on markup files without formatting them. The Oxygen DITA Map Metrics Report, for example, produces a statistical report instead of some flavor of formatted output.
3. They capture this formatted output in files – PDF or HTML5 typically.

Ultimately, the SimpliVity publishing environment has considerable flexibility in the vocabularies, DTDs, and processors that its uses to encode and to process content. We have no flexibility in changing XML grammar.

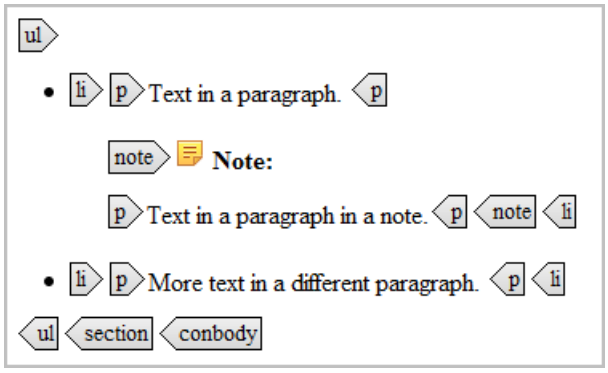
About Hierarchy, Inheritance, and Precedence

Although XML implementations share similar notions of grammar and vocabulary, they part company when it comes to how they implement hierarchy, inheritance, and precedence.

Hierarchy

Any element that contains another element is in a hierarchy of sorts. At a basic level, consider block elements such as tables, lists, or sections that can contain subordinate elements such as rows, columns, list items, or paragraphs. At a more sophisticated level, consider multiple levels of DITA maps. All elements in a hierarchical structure are, relative to one another, considered parents, children, or siblings (peers).

The following unordered list (<ul>) in XML DITA illustrates a simple containment hierarchy.



The two <li> elements under <ul> are direct child elements. The <p> and <note> elements within those child <li> elements are direct children of their respective <li> elements and indirect children of <ul>.

In XML DITA, these hierarchical blocks have two uses.

- You can assign them an @id and transclude (reuse by reference) the hierarchical blocks anywhere.
- You can address (query) the hierarchical structure using XPath (XQuery) as though it were a database structure. The following snippet specifies that XQuery examine the <p> element within the <note> which is within the <li> which is within the <ul>: `//ul/li/note/p`. In highly structured blocks in API reference topics or glossary items, you can extract and reuse values based on their position within the hierarchy.

The same logic for hierarchy applies to any XML DITA hierarchy – root maps, child maps, topics, and blocks.

Inheritance

Inheritance extends hierarchy by allowing attribute values specified for parent element to cascade automatically to all subordinate child elements in its hierarchical block. In the following example, the value "internal" for the attribute @audience is specified for the parent block element <ul>.

```
<ul audience="internal">
```

```
  <li>
    <p>Text in a paragraph. </p>
      <note>
        <p>Text in a paragraph in a note.</p>
      </note>
  </li>
  <li>
    <p>More text in a different paragraph. </p>
  </li>
</ul>
```

DITA inheritance rules silently apply all attribute values defined for a parent element to all direct and indirect child elements as though you added the inherited attribute to each child element manually.

```
<ul audience="internal">
  <li audience="internal">
    <p audience="internal">Text in a paragraph. </p>
      <note audience="internal">
        <p audience="internal">Text in a paragraph in a note.</p>
      </note>
  </li>
  <li audience="internal">
    <p audience="internal">More text in a different paragraph. </p>
  </li>
</ul>
```

The same logic for inheritance applies to any XML DITA hierarchy – root maps, child maps, topics, and blocks.

Precedence

Precedence rules define the order in which parsers resolve attribute values when there is more than one value for the same attribute defined within a hierarchical block. In the following example, note that there are now two values defined for @audience, "internal" and "infoeng".

```
<ul audience="internal">
  <li>
    <p>Text in a paragraph. </p>
      <note>
        <p>Text in a paragraph in a note.</p>
      </note>
  </li>
  <li audience="infoeng">
    <p>More text in a different paragraph. </p>
  </li>
</ul>
```

In XML DITA and any other object-oriented frameworks, the value assigned to an attribute in a parent element in the hierarchy is inherited by all child elements and takes precedence (gets used by parsers) over any alternate value assigned to the attribute. The value "internal" for @audience defined for the `<ul>` parent element takes precedence over the value "infoeng" defined for @audience attribute at a lower level. Both the value "internal" and the value "infoeng" get inherited by subordinate elements, but only "internal" gets used by parsers.

The same logic for precedence applies to any XML DITA hierarchy – root maps, child maps, topics, and blocks.

To determine which attribute values apply to which elements in a XML DITA structure, you need to understand the logic of its hierarchy, inheritance, and precedence.

About Static and Dynamic Assembly

The principles that we use to design and assemble technical documentation are changing dramatically.

Static assembly

Whether the building blocks for our publications are linear chapters or modular topics, most authoring tools require us to define some "assembly" for these building blocks. Whether that assembly definition is a Framemaker book, Flare TOC, or DITA map, someone needs to define a default sequence or hierarchy for assembling the building blocks. The DITA Open Toolkit, for example, requires that you provide the name of a map with links to topics before it will process any of your topic content. That classic "outline" or "bill of materials" represents but one static view of how the building blocks might be assembled. We define static DITA maps for our deliverables and present them to our customers as an "authoritative" view of the content. From a legal standpoint, the company has to stand by some definition of the functionality of its product, so some "authoritative" view is needed.

Does that "authoritative view" need to be the only view? Alternative ways to assemble the topic are never far away – just consult with product owners, scrum teams members, or Support professionals in your organization. Although most organizations are not yet in a position to leave static assembly behind, we cannot ignore the alternative method of organizing information – metadata-driven dynamic assembly.

Metadata-driven dynamic assembly

Two familiar technologies are nudging us away from relying exclusively on statically assembled content.

- *Social networking and analytics* – communities of users provide an alternate framework for evaluating the relative importance of the content we produce. The collective "like" or "dislike" pathways through technical documentation sets represent an alternate "authority" to the static table of contents. James Surowiecki argues in *The Wisdom of Crowds* that collective experience of large numbers of readers can be captured and modeled in real time. This experience, whether gathered interactively through social media or indirectly through analytics, offers an alternate view through the content that is often more relevant than statically assembled publications. Imagine some view of our topics called "Sort by audience and popularity."

  In *Too Big to Know*, David Weinberger writes --

  > The new filtering techniques are disruptive, especially when it comes to the authority of knowledge. Old knowledge institutions like newspapers, encyclopedias, and textbooks got much of their authority from the fact that they filtered information for the rest of us. If our social networks are our new filters, then authority is shifting from experts in faraway offices to the network of people we know, like, and respect.

- *Machine-based information processing* – the ability of crawlers, infobots, avatars, and search engines to distill vast quantities of technical information into non-sequential, non-hierarchical data structures is our current reality. The Googles and Pinterests serve as knowledge brokers, allowing individual readers to define their own priorities and pathways through the information. To a crawler, all content is metadata that it can process. The less meaningful metadata embedded in our content, the more the crawlers have to make guesses and fill in the blanks. The only issue for content developers is whether we know how to enrich the breadth and precision of the metadata in our content in such a way that machine processors build a more meaningful, useful picture of our content.

Designing your product documentation for dynamic assembly involves enriching the individual topics with meaningful metadata and semantic markup. Providing meaningful `<prolog>` information, `<abstract>` information, and accurate semantic tagging feeds the machine-based information processors with high-octane metadata and increases the likelihood that the topics served up to customers by these search systems will more accurately reflect what the customer is looking for. We need to trade control over the presentation of multiple topics (static assembly) for influence over the fidelity of content within any topic. In *Content Everywhere*, Sara Wachter-Boettcher writes –

> [It means] creating a new framework: a mindset focused on designing for and building content that can have multiple purposes, and whose meaning can stay intact through multiple contexts. It's about structuring content so that it's flexible enough to fit the varied needs that we have today, and strong enough to build on it with additional rules as they become necessary.

DITA – especially in the context of HTML5 delivery and content management metadata – is capable of supporting both statically defined and dynamic assembly. Investing some time in looking at alternatives to statically defined, map-driven deliverables is worth the effort.

About Information Typing and Semantic Markup

Information typing and semantic markup are the secret sauce for enterprise-class frameworks such as OASIS DocBook, OASIS DITA, and S1000D.

- *Information typing* – XML vocabularies that encourage writers to specify the types of information that they are developing – for example concepts, tasks, or reference topics – tend to produce more focused and more consistent writing. When there are many eyes in a writing organization looking at topics and sharing expectations for what a formal task ought to look like, you tend to get more consistent, more focused task writing. DITA topic types do not guarantee this sort of improved focus, but they can certainly encourage the development of shared expectations about information types.
- *Semantic markup* – lightweight markup languages tend to use output formatting styles such as boldface or italics or monospace to differentiate many different things in the source. In MS Word, for example, a monospace character style may be used by writers to indicate a filename, a filepath, a code snippet, or a command name. In OASIS DocBook or DITA, the writer chooses different semantic elements to perform the same differentiation – `<filepath>`, `<msgph>`, or `<cmdname>`. Old-school XML processors such as PDF may map these semantic elements to the same output styling (monospace), but the differentiated elements in the XML sources are now available to more savvy processors such as WebHelp or to metadata processors. Semantic tags can function analogously to database field names. Processors that can "query" these semantic elements can also assemble them dynamically into new topics such as API quick reference tables or CLI argument reference tables. Using @conrefs or @keyrefs to build a terminology tables from elements stored in disparate DITA glossary topics in commonplace. Each DITA glossterm topic is a database record waiting to be queried.

About Linear and Modular Authoring Environments

For a variety of reasons, the technical publishing industry does not have crisp, generally accepted set of definitions for linear and modular authoring environments. That said, the following definitions will suffice.

Linear authoring environments

Linear authoring environments consist of markup languages, authoring tools, and processors optimized for the development of publications designed to be read progressively (chapter-to-chapter, page-to-page, section-to-section). Distinguishing features include the following.

- Sophisticated stylesheet and layout logic.
- Sophisticated book-oriented assembly logic, e.g. prelims, part separators, chapter separators, et al..
- Style-driven, hierarchical organization, e.g. dedicated elements/styes for each heading level.

Some of the premier linear authoring environments for the technical publishing market include unstructured Framemaker, MS Word, and OASIS DocBook 4.x.

Modular authoring environments

Modular authoring environments consist of markup languages, authoring tools, and processors optimized for the development of content repositories full of reusable information components. Publications built from these reusable components can be organized statically (with hardwired maps) or dynamically (with metadata-driven processors). Distinguishing features include the following.

- Atomic units of authoring, e.g. topics, articles, or modules.
- Sophisticated logic for assembling these units, e.g. DITA maps, Flare TOCs, DocBook books, etc..
- Sophisticated logic for transcluding (reusing by reference) content at any level – words, sections, topic, and maps.
- Non-hierarchical organization, e.g. no level-specific headings or sections embedded in the markup.

- Separation of source content from delivered presentation. The same set of modules and/or maps can generate multiple deliverables, each with its distinct presentation format.

Some of the premier modular authoring environments for the technical publishing market include XML DITA, MadCap Flare, and OASIS DocBook 5.x.

There are no absolutes. If you apply best practices for modular writing to a linear authoring environment such as MS Word or unstructured Framemaker, you can produce some robust modular help systems. Similarly, authors in a modular environment can assemble and link modules into respectable linear manuals. That said, there is more logic and tooling in modular environments to produce linear documentation than there is logic and tooling in the linear environment to produce robust modular documentation.

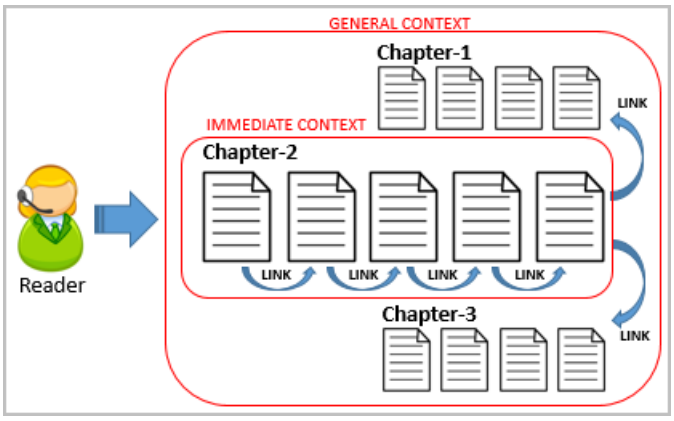
About Linear and Modular Documentation

Teams working in a modular authoring environment such as XML DITA can produce either linear and modular documentation. The differences between them can be focused externally (how people consume content) or internally (how content developers design and deliver content).

Content consumption

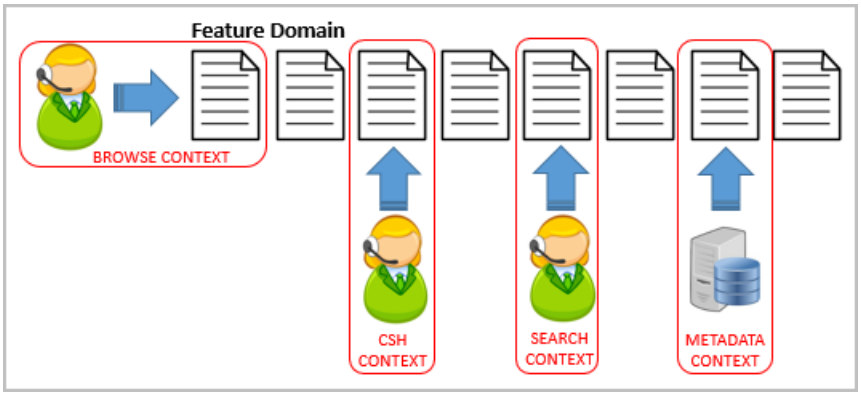
The following differentiators focus on how readers access linear or modular information.

Reader access frequency	If a deliverable is intended to be read (accessed) once for a release, it is unlikely that a customer would need to revisit specific topics in that deliverable. Deliverables focused on one-time use are more often than not linear. Deliverables such as software installation/deployment guides, upgrade guides, hardware installation guides, getting started guides, or release notes fit into this category. If a customer consults a deliverable while solving multiple, separate problems throughout a release cycle, then the topics in that deliverable should be candidates for modular development.
Reader access context	If the topics in a deliverable are designed to be read (accessed) by humans sequentially within the immediate context of a chapter and within the general context of a deliverable, those topics are closely associated with both the immediate and general reader context. By default, they live in a linear workflow.



Techniques that enhance the logic of the linear workflow – transitions, cross-references, section numbering – are beneficial and, in many cases, necessary.

If human or machine readers access specific topics from multiple, non-linear contexts – CSH, search, avatars – the default linear workflow is broken. These topics need to be developed as context-independent modules.



Content design and development

If we worked in a world that was not constrained by time or release cycles or staffing, we would not need to worry about the following distinctions. The more constrained our capacity, the more important these become.

Technical update frequency	If you need to update the majority of topics for a deliverable with every release, it is probably linear. Deliverables such as software installation/deployment guides, upgrade guides, hardware installation guides, getting started guides, or release notes fit into this category. Savings from content reuse across releases would be pretty minimal. If there is no technical reason to update the majority of topics for every release, that content may well be predominantly concepts or reference material. Insulating topics that rarely require technical change from their surrounding context (peer topics) makes sense. Needing to make non-technical updates to topics purely for the sake of updating contextual links can be expensive.
Content reuse	Topics that are reused in multiple contexts cannot contain context-specific cross-references, related links, or transitions. Product-specific content that can be filtered out at built time is OK, but not single-context cross-references, related links, or transitions. It is common for writing teams to increase the percentage of modular documentation over time as they discover opportunities for reuse. If writers discover that a topic in one linear "Getting Started" deliverable could be used in another linear "Getting Started" deliverable, then the newly shared topic should be refactored into a modular topic.
Content flexibility	Anyone who has built a stone wall can tell you that the *very* last thing that you add is the mortar between the individual stones. As the shape and constitution of the stone wall matures, it's likely you'll need to move things around or even tear down sections. By analogy, preserving modular, context-independent topics as along as possible in the design cycle makes sense. Product Managers, Product Owners, and scrum teams may benefit from seeing two or three alternate organizations (maps) for the content that we draft. Being able to reorganize the sequence and/or hierarchy of our topic content is a major benefit for a startup still trying to understand what it is we are building.

In many cases, one deliverable may require both linear and modular development. API guides can be a good example. The installation, configuration, and "getting started" sections are most effectively developed as linear docs. The subsequent feature overviews, concepts, how-to writing, and reference sections that may be relevant to other publications are modular.

About Encapsulation

In object-oriented software design, an object is considered "encapsulated" when it contains all the resources (data and methods) that it needs to function. Basically, it has no dependencies on other objects for its primary operation.

For XML DITA objects (topics) to function as independent modules, they must not include the typical XML structures that create topic-to-topic dependencies.

- `<xref>`s
- `<related-links>` and associated `<link>`s
- Transcluded content via @conrefs

In the transition from developing linear documentation to modular documentation, encapsulation is probably the most challenging and rewarding concept.

Topics under contruction

About information repositories, builds, and deliverables

TBD

About progressive discloure

TBD

About user assistance design and integration

TBD

About scaling information architecture

Scaling is growing an infrastructure (compute, storage, networking) larger so that the applications riding on that infrastructure can serve more people at a time.

Scale up

Scaling up is taking what you've got, and replacing it with something more powerful. From a networking perspective, this could be taking a 1GbE switch, and replacing it with a 10GbE switch. Same number of switchports, but the bandwidth has been scaled up via bigger pipes.

At first, adding more disks improves performance because disk throughput was probably the limiting factor from a performance standpoint. However, as the load on the array climbed — a situation often driven by virtualization — and more disks were added, the two controllers themselves became a bottleneck as each began to require more and more CPU for RAID calculations. Eventually, enough disks were added that the controllers were simply saturated and could do no more. Adding more and faster disks behind an overloaded controller pair simply places more overload on the controllers. In these systems once the controllers are the bottleneck there is little you can do, apart from buying an additional new array with its own pair of controllers and moving some of the workload (VMs) onto the new array.

Scale out

Scaling out takes the infrastructure you've got, and replicates it to work in parallel. This has the effect of increasing infrastructure capacity roughly linearly. Data centers often scale out using pods. Build a compute pod, spin up applications to use it, then scale out by building another pod to add capacity. Actual application performance may not be linear, as application architectures must be written to work effectively in a scale-out environment.

Each server is loaded with disks and uses a network — usually Ethernet — to talk to the other servers in the storage array. The group of servers together forms a clustered storage array and provides LUNs or file shares over a network just like a traditional array. Adding additional nodes of disks to a scale out array actually adds another x86 server

to the cluster. At the same time, doing so adds more network ports, more CPU and more RAM. As the capacity of a scale-out array increases so does its performance, adding nodes usually results in linear performance improvements. Adding nodes is generally non-disruptive, so a normal maintenance window can be used to add capacity and performance to the array, rather than a full system outage.

Scale through