

*Best Practices for Creating
Universally Usable Content*

**Excerpt from
EPUB 3 Best Practices**

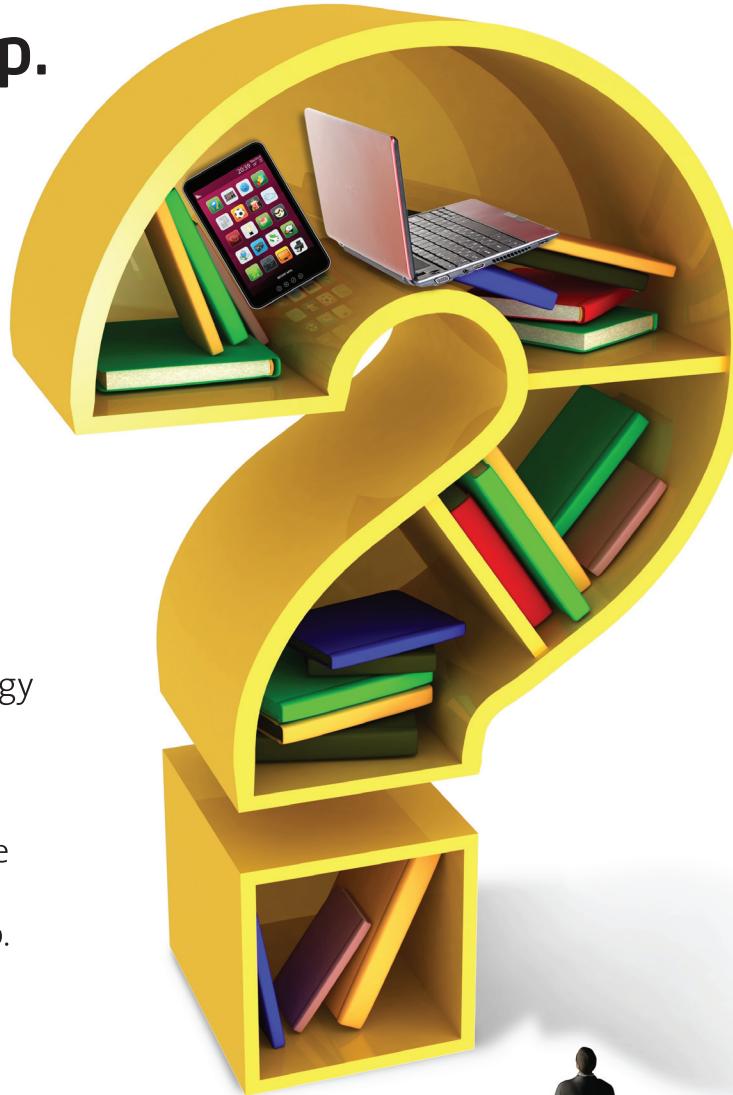


Accessible EPUB 3

O'REILLY®

Matt Garrish

Having trouble in the digital world? SPi Global can help.



eBooks, ePub and HTML5 – technology is changing at a dizzying pace. If you are having a challenge keeping up with how technology can help you maximize your content or don't have the resources to effectively leverage the digital space, SPi Global can help.

For more information, please email jamie.israel@spi-global.com or call 732 662 8345.



Accessible EPUB 3

Matt Garrish

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Accessible EPUB 3

by Matt Garrish

Copyright © 2012 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://my.safaribooksonline.com>). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

Editor: Brian Sawyer

Production Editor: Dan Fauxsmith

Proofreader: O'Reilly Production Services

Cover Designer: Karen Montgomery

Interior Designer: David Futato

Illustrator: Robert Romano

Revision History for the First Edition:

2012-02-07 First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781449328030> for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Accessible EPUB 3* and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-32803-0

[LSI]

1328628844

Table of Contents

Preface	v
1. Introduction	1
The Digital Famine	2
Accessibility and Usability	3
2. Building a Better EPUB: Fundamental Accessibility	7
A Solid Foundation: Structure and Semantics	7
Data Integrity	9
Separation of Style	11
Semantic Inflection	12
Language	13
Logical Reading Order	14
Sections and Headings	15
Context Changes	17
Lists	18
Tables	19
Figures	22
Images	22
SVG	23
MathML	24
Footnotes	26
Page Numbering	26
Getting Around: Navigating an EPUB	27
The Untold Story: Metadata	32
3. It's Alive: Rich Content Accessibility	37
The Sound and the Fury: Audio and Video	37
Timed Tracks	39
Talk to Me: Media Overlays	41
Building an Overlay	44

Structural Considerations	49
Tell It Like It Is: Text-to-Speech (TTS)	50
PLS Lexicons	53
SSML	57
CSS3 Speech	61
The Coded Word: Scripted Interactivity	65
A Little Help: WAI-ARIA	67
Custom Controls	67
Forms	73
Live Regions	75
A Blank Slate: Canvas	77
4. Conclusion	79
EPUB 3 Best Practices Teaser	79
About the Book	81

Preface

Accessibility is a difficult concept to define. There's no single magic bullet solution that will make all content accessible to all people. Perhaps that's a strange way to preface a book on accessible practices, but it's also a reality you need to be aware of. Accessible practices change, technologies evolve to solve stubborn problems, and the world becomes a more accessible place all the time.

But although there are best practices that everyone should be following, and that will be detailed as we go along, this guide should neither be read as an instrument for accessibility compliance nor as a replacement for existing guidelines.

The goal is to provide you with insights and ideas into how to begin making your publications richer for all readers at the same time that you make them more accessible. Proliferating usability guidelines and muddying the waters of compliance is not its intent. There are areas that would take a book unto themselves to explore in detail in relation to the use of HTML5 content within EPUB, such as the Web Content Accessibility Guidelines (WCAG) and Web Accessibility Initiative's Accessible Rich Internet Applications (WAI-ARIA). Whenever issues extend beyond what can be covered in these best practices, pointers to where you can obtain more information will be included. Don't fall into the trap of hand-picking accessibility.

It is also naturally the case with a standard as new and wide-ranging as EPUB 3 that best practices will evolve and develop as the features it offers are explored and implemented. This guide will endeavor to make clear whenever uncertainty exists around an approach, what alternatives there are, and where you should be looking to watch for developments.

You need to be thinking about accessibility and planning good content practices from the outset if you're going to make the most of the features EPUB 3 has to offer. This guide will be your map, but you have to be willing to follow it.



This guide is envisioned as a living document and intended to be updated and re-released as new practices and techniques evolve.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Accessible EPUB 3* by Matt Garrish (O'Reilly). Copyright 2012 Matt Garrish, 9781449328030."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

Safari® Books Online



Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at <http://my.safaribooksonline.com>.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

<http://shop.oreilly.com/product/0636920025283.do>

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

I would like to thank the following people for their invaluable input, assistance reviewing, and plain general patience answering my dumb questions along the way: Markus Gylling, George Kerscher, Daniel Weck, Romain Deltour, and Marisa DeMiglio from the DAISY Consortium; Graham Bell from EDItEUR; Dave Gunn from RNIB; Ping Mei Law, Richard Wilson, and Sean Brooks from CNIB; and Dave Cramer from Hachette Book Group.

And a special second thanks to Markus, Bill McCoy, and George for the opportunity I was given to be involved in the EPUB revision and to write this guide.

And a final thanks to Brian Sawyer and all the people at O'Reilly for their work putting this guide together!

Introduction

If you're expecting a run-of-the-mill best practices manual, be aware that there's an ulterior message that will be running through this one. While the primary goal is certainly to give you the information you need to create accessible EPUB 3 publications, it also seeks to address the question of *why* you need to pay attention to the quality of your data, and how accessible data and general good data practices are more tightly entwined than you might think.

Accessibility is not a feel-good consideration that can be deferred to republishers to fill in for you as you focus on print and quick-and-dirty ebooks, but a content imperative vital to your survival in the digital future, as I'll take the odd detour from the planned route to point out. Your data matters, not just its presentation, and the more you see the value in it the more sense it will make to build in accessibility from the ground up.

It's a common misconception, for example, that any kind of data is accessible data, and that assistive technologies like screen readers work magic and absolve you of paying attention to what's going on "under the hood," so to speak. Getting the message out early that this is not the case is essential to making EPUB more than just a minimally accessible format and preventing past mistakes from being perpetuated.

It's unfortunately too easy when moving from a visual medium like print to treat digital content as nothing more than yet another display medium, however. The simple path is to graft what you know onto what you don't. But it's that thinking that perpetuates the inaccessibility of content. Everything starts with the source. All the bells and whistles your reading system can do for you to assist in rendering and playback ultimately rely on the value of the content underneath and the ability to make sense of it.

Treat your data as a second-class citizen and eventually you'll be recognized as a second-class publisher.

But try and turn your brain off to the word accessibility as you read this guide and focus instead on the need to create rich, flexible, and versatile content that can make the reading experience better for everyone.

Inaccessible content typically means you're settling for the least value you can get, so get ready to think bigger.

The Digital Famine

Before getting into the best practices themselves, there are two subjects that it would be a lapse for me to not talk about first. The digital famine is the first, as it will hopefully give you some real-world perspective on why accessibility matters.

You're probably wondering what the famine is, since there are some impressive statistics emerging to show that the ebook revolution isn't slowing down any time soon. Unfortunately, the numbers aren't where it matters most yet if you believe in universal access to information. Sales are rising exponentially year over year, but the number of accessible ebooks available at the source is still small.

A commonly cited statistic in accessibility circles is that only about 5 percent of the books produced in any year are ever made available in an accessible format. Although there are signs that this rate is beginning to tick upward with more ebooks being produced, the overall percentage of books that become available in accessible formats still remains abysmally small. Fiction bestsellers are a bright spot, as they've been the first to receive the digital treatment, but there's more to reading than just fiction.

Picture yourself in the situation where you'll only ever have a spattering of books at your fingertips in any given subject area, and probably none in the more niche topics you delve into. It's not a matter of finding another bookstore or reading application; those books just aren't coming and there's nothing you can do to change it. This dearth of content is what people refer to as the *digital famine*.

Not a pleasant thought, and it's a reality that many people are forced to live right now; it's only imaginary if you're fortunate not to be affected. The ebook revolution holds out the promise of improvement, as mainstream publishing finds itself suddenly charting the same path as accessible producers, but there are still a number of factors that will contribute to this paltry number for some time to come, including:

- New workflows haven't yet emerged to facilitate the transition. Mass retail ebook production and consumption took many people by surprise, the author included, after earlier failed attempts. Tools and production systems are not optimized for high-quality multi-stream output production, making internal conversion of print to digital costly.
- Accessible ebooks can become inaccessible after ingestion into a distribution channel, whether via reformatting to less feature-rich formats or for feature-reduced reading.
- The inaccessibility of online bookstores themselves can hinder the ability to obtain ebooks.

- Libraries for the blind and other republishers don't have the resources to completely re-engineer the print-only books still being produced. And this model is a failing one for the long-term ideal of full content accessibility.

But, while depressing in the short term, none of these issues are insurmountable, and none are antithetical to producing good content. It's only to say that there are interesting times ahead, and to reinforce that there remains much still to be done. The existence of EPUB 3 alone does not cure this famine.

Accessibility and Usability

The other subject that needs treatment is what is meant by *accessibility* and *usability* in the context of this guide. These two terms are often used in overlapping fashion, and can mean different things to different people, but I'll be using the following definitions:

Accessibility of content is the intrinsic capabilities of the EPUB 3 publication: the quality of the data and meaning that can be extracted from it; the built-in navigational capabilities; the additional functionality, like text and audio synchronization (media overlays) and improved synthetic speech. The publisher of an EPUB has control over the accessibility of their publication, whether directly through the tools they use to generate the source or in post-production workflows.

Usability is the ability of a reader to access the content on any given reading system. A publisher may make an EPUB 3 publication rich with accessibility features, but if a reader does not have the right device or software program to access those features it is not the publication itself that is to blame.

But even making these distinctions, there's no simple answer to what a fully accessible EPUB is, or to what a completely usable reading system is. It means something different depending on your needs.

A person with a print disability, for example, "cannot effectively read print because of a visual, physical, perceptual, developmental, cognitive, or learning disability" ([DAISY Glossary](#)). The best method to address any one of these areas is not necessarily the best method to address any of the others. Audio is necessary for readers who are blind, for example, but a reader who is dyslexic might benefit from audio, or from font changes or visual cues, or from a combination of these. There's no universal answer.

And with EPUB 3 opening the door to new rich multimedia experiences, so too do you need to think beyond traditional print disabilities and recognize that ebooks have the potential to exclude a greater segment of the population if not done with care:

- the inability to hear embedded audio and video is a concern for persons who are deaf or hard of hearing;
- interactivity and animations that rely on color recognition have the potential to exclude persons who are color blind or have difficulty distinguishing blended contrasts;

- the new trend to voice activated devices has the potential to make reading for persons with speech impairments difficult.

The point isn't to suggest that the problem is too big to try and tackle, in fact the opposite. If you haven't caught on, I'm making the case why ignoring accessibility means ignoring a large segment of readers who would love to be buying and consuming your ebooks. It is estimated that 10 percent of the population has a print disability; that's a large market you could be catering to to increase your sales.

And we haven't yet touched on *situational disabilities*. A situational disability is one in which a person who would otherwise be able to interact with your ebook is in a position in which they can't, or find themselves facing the same limitations. For example:

- someone trying to read on a cell phone will gain an appreciation for the difficulty of reading small sections of prose at a time, as someone with low vision experiences when reading using zooming software;
- someone attempting to read on their deck on a bright summer day, angling and holding their tablet close to their face to follow the prose, will understand the difficulty experienced by someone with age-related sight loss and/or who has trouble with contrasts;
- someone sitting on the subway going home who has to turn on subtitles in an embedded video to read the dialogue will experience how a person who is deaf interacts with the video.

In other words, everyone will benefit from accessible data at some point in their lives, as there are a lot of ways accessible data improves access that aren't always immediately obvious. Accessibility is critical for some and universally beneficial for all.

The richer you make your data the more intelligently it can be used; so even though you may not be able to accommodate everyone at the end of the day, you can go a long way toward accommodating the majority with a number of simple measures. And that is the focus of this guide.

Usability as defined here, however, is outside the realm of content production, and can't be tackled by a guide whose focus is increasing the quality of your content. The EPUB specification bakes in some requirements on the reading system side to improve usability, but not every reading system is going to support every accessible feature, and usability is not just support for EPUB but extends into the design of reading systems themselves.

You can't let usability influence your accessibility decision making, however. A typical practice is to target the industry-leading platform and build around its capabilities (and deficiencies), but what value does this bring you in the long term? Think of the cost that resulted from making Internet Explorer-only friendly websites when it held 90 plus percent of the market as an example of where following the leader can take you. Your books will hopefully be selling well for years to come, but unless you enjoy reformatting

from scratch each time you look to upgrade or enhance, it pays to put the effort into doing them right up front.

But it's time to roll up our sleeves and get our hands dirty...

Building a Better EPUB: Fundamental Accessibility

This guide takes a slightly different approach to accessibility because of the feature-rich nature of EPUB 3. Instead of grouping all the practices together under a single rubric of essentiality, I'm going to instead take a two-tier approach to making your content accessible.

This first section deals with the core text and image EPUB basics, while the second ventures into the wilder areas, like scripting and the new accessible superstructures you can build on top.

I'm going to start with a section on the fundamentals of accessible content, naturally enough, because if you get your foundation wrong, everything else degrades along with it.

A Solid Foundation: Structure and Semantics

The way to begin a discussion on the importance of structure and semantics is not by jumping into a series of seemingly detached best practices for markup, but to stop for a moment to understand what these terms actually mean and why they're so important to making data accessible. We'll get to the guidelines soon enough, but if you don't know why structure and semantics matter, you're already on the fast track to falling into the kinds of bad habits that make digital data inaccessible, no matter the format.

Although the terms are fairly ubiquitous when it comes to discussing markup languages and data modeling generally—because they are so important to the quality of your data and your ability to do fantastic-seeming things with it—they are often bandied about in ways that make them sound geeky and inaccessible to all but data architects. I'm going to try and make them more accessible in showing how they facilitate reading for everyone, however.

Let's start simple, though. You're probably used to hearing the terms defined along these lines: *structure* is the elements you use to craft your EPUB content, and *semantics* is the additional meaning you can layer on top of those structures to better indicate what they represent.

But that's undoubtedly a bit esoteric if you don't go mucking around in your markup on a regular basis, so let's take a more descriptive approach to their meaning. Another way to think about their importance and relationship is via a little reformulation of Plato's allegory of the cave. In this dialogue, if you've forgotten your undergrad Greek philosophy, Socrates describes how the prisoners in the cave can only see shadows of the true forms of things on the walls as they pass in front of a fire, and only the philosopher kings will eventually break free of the chains that bind them in ignorance and come to see the reality of those forms.

As we reformulate Plato, the concept of generalized and specific forms is all that you need to take away from the original allegory, as getting from generalized to specific is the key to semantic markup. In the new content world view I'm proposing, the elements you use to mark up a document represent the generalized reflection of the reality you are trying to express. At the shadow level, so to speak, a chapter and a part and an introduction and an epilogue and many other structures in a book all function in the same way, like encapsulated containers of structurally significant content.

These general forms allow markup grammars, like HTML5, to be created without element counts in the thousands to address every possible need. A generalized element retains the form of greatest applicability at the expense of specifics, in other words. The HTML5 grammar, for example, solves the problem of a multitude of structural containers with only slightly differing purposes by introducing the `section` element.

But what help is generalized markup to a person, let alone a reading system, let alone to an assistive technology trying to use the markup to facilitate reading? Try making sense of a markup file by reading just the element names and see how far you get; a reading system isn't going to fare any better despite a developer's best efforts. HTML5 may now allow you to group related content in a `section` element, for example, but without reading the prose for clues all you know is that you've encountered a seemingly random group of content called section. This is structure without semantics.

You might think to make out the importance of the content by sneaking a peek ahead at the section's heading—assuming it has one—but unless the heading contains some keyword like “part” or “chapter” you still won't know why the section was added or how the content is important to the ebook as a whole. And cheating really isn't fair, as making applications perform heuristic tests like looking at content can be no small challenge. This is both the power and failing of trying to process generalized markup languages and do meaningful things with what you find: you don't have to account for a lot, but you also don't often get a lot to work with.

Getting back to our analogy, though, it's fair to say we're all philosopher kings when it comes to the true nature of books; we aren't typically interested in, and don't typically

notice, generalized forms when reading. But, whether we realize it or not, we rely on our reading systems being able to make sense of these structures to facilitate our reading, and much more so when deprived of sensory interactions with the device and content. When ebooks contain only generalized structures, reading systems are limited to presenting only the basic visual form of the book. Dumb data makes for dumb reading experiences, as reading systems cannot play the necessary role of facilitator when given little-to-nothing to work with. And that's why not everyone can read all digital content.

It's not always obvious to sighted readers at this point why semantics are important for them, though, as they just expect to see the visual presentation the forms provide and to navigate around with fingers and eyes. But that's also because no one yet expects more from their digital reading experience than what they were accustomed to in print. Knowing whether a section is a chapter or a part as you skip forward through your ebook can make it so you don't always have to rely on opening the table of contents. Knowing where the body matter section begins can allow a reading system to jump you immediately to the beginning of the story instead of the first page of front matter. Knowing where the body ends and back matter begins could allow the reading system to provide the option to close the ebook and go back to your bookshelf; it might also allow links to related titles you might be interested in reading next to be displayed. Without semantically rich data, only the most rudimentary actions are possible. With it, the possibilities for all readers are endless.

So, to wrap up the analogy, while some of us can read in the shadow world of generalized markup, all we get when we aim that low is an experience that pales to what it could be, and one that needlessly introduces barriers to access. If I've succeeded in bringing these terms into relief, you can hopefully now appreciate better why semantics and structure have to be applied in harmony to get the most value from your data. The accessibility of your ebook is very much a reflection of the effort you put into it. The reading system may be where the magic unfolds for the reader, but all data magic starts with the quality of the source.

With that bit of high-level knowledge under our belts, let's now turn to how the two work together in practice in EPUB 3 to make content richer and more accessible.

Data Integrity

The most important rule to remember when structuring your content is to use the right element for the right job. It seems like an obvious statement, but too often people settle for the quick solution without thinking about its impact; look no further than the Web for examples of markup run amok. Print to digital exports are also notorious for taking the path of least complexity (p-soup, as I like to call the output that wraps most everything in paragraph tags). In fairness, though, print layout programs typically lack the information necessary for the export to be anything more than rudimentary.

When present, however, reading systems and assistive technologies are able to take advantage of specialized tags to do the right thing for you, but there's little they can do if you don't give them any sense of what they're encountering.

When it comes to EPUB 3, if you don't know what's changed in the new HTML specification, go and read the element definitions through; it's worth the time. EPUB 3 uses the XHTML flavor of HTML5 for expressing text content documents, so knowledge of the specification is critical to creating good data. Don't assume knowledge from HTML4, as the purpose of many elements has changed, and elements you thought you knew might have different semantic meanings now (especially the old inline formatting elements like `i`, `b`, `small`, etc.).

And remember that structure is not about what you want an element to mean. The changes to the HTML5 element definitions may not always make the most sense (see the human restriction on the `cite` element as one commonly cited example), but twisting definitions and uses to fit your own desires isn't going to make you a friend of accessibility, either. Reading systems and assistive technologies are developed around the common conventions.

And whatever you do, don't perpetuate the sin of immediately wrapping `div` and `span` tags around any content you don't know how to handle. It's a [violation](#) of the EPUB 3 specification to create content that uses generic elements in place of more specific ones, and it doesn't take long to check if there really is no other alternative first. When you make up your own structures using generic tags, you push the logical navigation and comprehension of those custom structures onto the reader (and potentially mess up the HTML5 outline used for navigation). Sighted readers may not notice anything, but when reading flows through the markup, convoluted structures can frustrate the reader and interfere with their ability to effectively follow the narrative flow.

If you don't discover an existing element that fits your need, the process of checking will typically reveal that you're not alone in your problem, and that community-driven solutions have been developed. Standards and conventions are the friend of accessibility. And if you really don't know and can't find an answer, ask. The IDPF maintains [discussion forums](#) where you can seek assistance.

There are, of course, going to be many times when you have no choice but to use a generic tag, but when you do, always try to attach an `epub:type` attribute with a specific semantic (we'll cover this attribute in more detail shortly). The more information you can provide, the more useful your data will be.

Take the converse situation into consideration when creating your content, too. You aren't doing readers a service by finding more, and ever complex, ways to nest simple structures. The more layers you add the harder it can be to navigate, as I already mentioned. Over-analyzing your data can be as detrimental to navigation as under-analyzing.

For persons who cannot visually navigate your ebook, this basic effort to properly tag your data reduces many of the obstacles of the digital medium. The ability to skip structures and escape from them starts with meaningfully tagged data. The ability to move through a document without going to a table of contents starts with meaningfully tagged data.



Skipping and escaping are terms that will come up repeatedly in this guide. *Skipping*, as you might expect, is the ability to ignore elements completely, to skip by them. Accessible reading systems typically provide the ability for the reader to specify the constructs they wish to ignore, such as sidebars, notes, and page numbers. *Escapable* content typically consists of deep-nested or repetitive structures—such as found in tables and lists—that a user may wish to move out from in order to continue reading at the next available item following the escaped content (a reading system’s user interface would normally provide quick access to the “escape” command, so that the operation can easily be called repetitively, if needed).

The integrity of your data is also a basic value proposition. Do you expect to throw away your content and start over every time you need to re-issue, or do you want to retain it and be able to easily upgrade it over time? Structurally meaningful data is critical to the long-term archivability of your ebooks, the ability to easily enhance and release new versions as technology progresses, as well as your ability to interchange your data and use it to create other outputs. Start making bad data now and expect to be paying for your mistakes later.

Separation of Style

Some old lessons have to be continuously relearned and reinforced, and not mixing content and style is a familiar friend to revisit whenever talking about accessible data.

To be clear, separating style does not mean avoiding the `style` attribute and putting all your CSS in a separate file, even if that is another good practice we’ll get back to. What separation of style refers to is not expecting the visual appearance of your content to convey meaning to readers. Style is just a layer between your markup and the device that renders it, not an intrinsic quality you can rely on to say anything about your content. Typographic conventions had to convey meaning in print because that was all that was available, and are still useful for sighted readers, but are the wrong place now to be carrying meaning.

Some reading systems will give you the full power of CSS, while others won’t even have a screen for reading. Some readers will visually read your content, while others will be using nonvisual methods. If only the visual rendering of your content conveys meaning to the reader, you’re failing a major accessibility test. Leave style in that in-between

layer where it targets visual readers, and keep your focus on the quality of your markup so that everyone wins.

The most basic rule of thumb to remember is that if you remove the CSS from your ebook, you should still be able to extract the same meaning from it as though nothing had changed. Your markup is what should ultimately be conveying your meaning. If you rely solely on position or color or whatever other stylistic flair you might devise, you're taking away the ability of a segment of your readers to understand the content.

But there is something to be said for cleanly separating content from style at the file level, too. The cascading nature of styles means that the declaration closest to the element to be rendered wins. If you tack `style` attributes all over your content you can interfere with the ability of a reader to apply an alternate style sheet to improve the contrast, for example, or to change the color scheme, as the local definition may override the problem the reader is attempting to fix. Consequently, suggesting that you avoid the `style` attribute like the plague is actually not an overstatement.

More realistically, though, you should be able to use CSS classes for your needs. If, for some reason, you do have to add a `style` attribute, though, avoid using it to apply general stylistic formatting. Keeping your style definitions in a separate file simplifies their maintenance and facilitates their re-use on the production side, anyway, and this simple standard practice nets you an accessibility benefit.

Semantic Inflection

I'm not going to rehash the reasons for semantic markup again, but I intentionally neglected getting into the specifics of how they're added in EPUB 3 until now so as not to confuse the need with the technical details.

Adding semantic information to elements is actually quite simple to do; EPUB 3 includes the `epub:type` attribute for this purpose. You can attach this attribute to any HTML5 element so long as you declare the `epub` namespace. The following example uses the attribute to indicate that a `dl` element represents a glossary:

```
<html ... xmlns:epub="http://www.idpf.org/2007/ops">
...
  <dl epub:type="glossary">
    <dt><dfn>Brimstone</dfn></dt>
    <dd>Sulphur; See <a href="#def-sulphur">Sulphur</a>.</dd>
  </dl>
...
</html>
```

Whenever you use unprefixed values in the attribute (i.e., without a colon in the name), they must be defined in the [EPUB 3 Structural Semantics Vocabulary](#). All other values require a defined prefix and are typically expected to be drawn from industry-standard vocabularies. In other words, you cannot add random values to this attribute, like you can with the `class` attribute.

You can create your own prefix, however, and use it to devise any semantics you want, but don't create these kinds of custom semantics with the expectation they will have an effect on the accessibility or usability of your ebook. Reading systems ignore all semantics they don't understand and don't have built-in processing for. It would be better to work with the IDPF or other interested groups to create a vocabulary that meets your needs if you can't locate the semantics you need, as you're more likely to get reading system support that way.

The attribute is not limited to defining a single semantic, either. You can include a space-separated list of all the applicable semantics in the attribute.

A `section`, for example, often may have more than one semantic associated with it:

```
<section epub:type="toc backmatter">
  ...
</section>
```

The order in which you add semantics to the attribute does not infer importance or affect accessibility, so the above could have just as meaningfully been reversed.

You should also be aware that this attribute is only available to augment structures; it is not intended for semantic enrichment of your content. Associating the personal information about an individual contained in a book so that a complete picture of their life can be built by metadata querying, for example, is not yet possible. The metadata landscape was considered too unstable to pick a method for enriching data, but look for a future revision to include this ability, whether via RDFa, microdata, or another method.

And in case it needs repeating, semantics are not just an exercise in labeling elements. As I discussed in the introduction to this section, these semantics are what enable intelligent reading experiences. If you had 25 definition lists in an ebook each with a particular use, how would a reading system determine which one represents the glossary if you didn't have a semantic applied as in the first example? If you know which is the glossary, you could provide fast term lookups. The easier you make it for machines to analyze and process your data, the more valuable it becomes.

Language

Although the global language for the publication is set in the EPUB package file metadata, it's still a good practice to specify the language in each of your content documents. In an age of cloud readers, assistive technologies might not have access to the default language if you don't (unless they rewrite your content file to include the information in the package document, which is a bad assumption to make). Without the default language, you can impact on the ability of the assistive technology to properly render text-to-speech playback and on how refreshable braille displays render characters.

An `xml:lang` attribute on the root `html` element is all it takes to globally specify the language in XHTML content documents. For compatibility purposes, however, you should also include the HTML `lang` attribute. Both attributes must specify the same value when they're used.

We could indicate that a document is in German as follows:

```
<html ... xml:lang="de" lang="de">
```

Similarly, for SVG documents, we add the `xml:lang` attribute to indicate that the title, description, and other text elements are in French:

```
<svg ... xml:lang="fr">
```

You should also clearly identify any prose within your book that is in a different language from the publication:

```
<p>She had an infectious <i xml:lang="fr" lang="fr">joie de vivre</i> mixed with a certain <i xml:lang="fr" lang="fr">je ne sais quoi</i>.</p>
```

The `xml:lang` attribute can be attached to any element in your XHTML content documents (and the `lang` attribute is again included for compatibility). Properly indicating when language of words, phrases, and passages changes allows text-to-speech engines to voice the words in the correct language and apply the proper lexicon files, as we'll return to in more detail in the text-to-speech section.

Logical Reading Order

Although you'll hear that all EPUB 3s have a default reading order, it's not necessarily the same thing as the logical reading order, or primary narrative. The EPUB 3 `spine` element in the publication manifest defines the order in which a reading system should render content files as you move through the publication. This default order enables a seamless reading experience, even though the publication may be made up of many individual content files (e.g., one per chapter).

But although the main purpose of the `spine` is to identify the sequence in which documents are rendered, you can use it to distinguish primary from auxiliary content files. The `linear` attribute can be attached to the child `itemref` elements to indicate whether the referenced content file contains primary reading content or not. If a content file contains auxiliary material that would normally appear at the point of reference, but is not considered part of the main narrative, it should be indicated as such so that readers can choose whether to skip it.

For example, if you group all your chapter end notes in a separate content document, you could indicate their auxiliary status as follows:

```
<spine>
...
<itemref idref="chapter1"/>
<itemref idref="chapter1-notes" linear="no"/>
<itemref idref="chapter2"/>
```

```
<itemref idref="chapter2-notes" linear="no"/>
...
</spine>
```

A reader could now ignore these sections and continue following the primary narrative uninterrupted. But this capability is only a simple measure for distinguishing content that is primary at the macro level; it's not effective in terms of distinguishing the primary narrative flow of the content within any document. (Although in the case of simple works of fiction that contain only a single unbroken narrative, it might be.)

Sighted readers don't typically think about the logical reading order within the chapters and sections of a book, but that's because they can visually identify the secondary content and read around it as desired. A reading system, however, doesn't have this information to use for the same effect unless you add it (those semantics, again).

As I touched on in keeping style separate from content, you can, for example, give a sidebar a nice colorful border and offset it from the narrative visually using a `div` and CSS, but you've limited the information you're providing to only a select group when all you use is style. Using a `div` instead of an `aside` element means a reading system will not know by default that it can skip the sidebar if the reader has chosen to only follow the primary narrative.

For someone listening to the book using a text-to-speech engine, the narrative will be interrupted and playback of the sidebar `div` will be initiated when you mis-tag content in this way. The only solution at the reader's disposal might be to slowly move forward until they find the next paragraph that sounds like a continuation of what they were just listening to (`div` elements aren't always escapable). Picture trying to read and keep a thought with the constant interruptions that can result from sidebars, notes, warnings and all the various other peripheral text material a book might contain.

For this reason, you need to make sure to properly identify content that is not part of the primary narrative as such. The `aside` element is particularly useful when it comes to marking text that is not of primary importance, but even seemingly small steps like putting all images and figures in `figure` tags allows the reader to decide what additional information they want presented. I'll be returning to how to tag many of these as we go, too.

The [EPUB 3 Structural Semantics Vocabulary](#) is also a useful reference when it comes to which semantics and elements to apply to a given structure. Each of the semantics defined in this vocabulary indicates what HTML element(s) it is intended to be used in conjunction with.

Sections and Headings

As I touched on in the introduction to this section, always group related content that is structurally significant in `section` elements to facilitate navigation, and always indicate why you've created the grouping using the `epub:type` attribute:

```
<section epub:type="epilogue">
  ...
</section>
```

The entries in the table of contents in your navigation document are all going to be structurally significant, which can be a helpful guide when it comes to thinking about how to properly apply the `section` element. Some additional ideas on structural significance can be gleaned from the terms in the [EPUB 3 Structural Semantics Vocabulary](#). For example, a non-exhaustive list of semantics for sectioning content includes:

- foreword
- prologue
- preface
- part
- chapter
- epilogue
- bibliography
- glossary
- index

Semantics are especially helpful when a section does not have a heading. Sighted readers are used to the visual conventions that distinguish dedications, epigraphs, and other front matter that may be only of slight interest, for example, and can skip past them. Someone who can't see your content has to listen to it if you don't provide any additional information to assist them.

Headingless, unidentified content also means the person will have to listen to it long enough to figure out why it's even there. Have you just added an epigraph to the start of your book, and skipping the containing `section` will take them to the first chapter, or are they listening to an epigraph that starts the first chapter and skipping the `section` will take them to chapter two? These are the impediments you shift onto your reader when you don't take care of your data.

When the `section` does contain a heading, there are two options for tagging: numbered headings that reflect the current level or `h1` headings for every `section`. At this point in time, using numbered headings is recommended, as support for navigation via the structure of the document is still developing:

```
<section epub:type="part">
  <h1>Part I</h1>

  <section epub:type="chapter">
    <h2>Chapter 1</h2>
    ...
  </section>
</section>
```

Numbered headings will also work better for forward-compatibility with older EPUB reading systems.

Using an `h1` heading regardless of the nesting level of the section will undoubtedly gain traction moving forward, though. In this case, the `h1` becomes more of a generic heading, as traversal of the document will occur via the document outline and not by heading tags (the [construction of this outline](#) is defined in HTML5). There is only limited support for this method of navigation at this time, however.



It's also worth briefly noting that the `hgroup` element should probably only be used judiciously, if at all, for title and subtitle grouping at this time. The element is not yet widely supported, and there are a number of [proposals to change it](#) under review as of writing.

And remember that titles are an integral unit of information. If you want to break a title across multiple lines, change font sizes, or do other stylistic trickery, use spans and CSS and keep the display in the style layer. *Never* add multiple heading elements for each segment. Use `span` elements if you need to visually change the look and appearance of headings.

To break a heading across lines, we could use this markup:

```
<h1>Chapter <span class="chapNum">One</span> Loomings.</h1>
```

and then add the following CSS class to change the font size of the `span` and treat it as a block element (i.e., place the text on a separate line):

```
span.chapNum {  
  display: block;  
  margin: 0.5em 0em;  
  font-size: 80%  
}
```

If you fragment your data, you fragment the reading experience and cause confusion for someone trying to piece back together what heading(s) they've actually run into.

Context Changes

A nasty side-effect of current print-based export processes is that changes in context are visually styled either using CSS and/or with images. When you use the CSS `margin-top` property to add spacing, you're taking away from anyone who can't see the display that a change in context has occurred. Graphics to add whitespace are no better, since they don't typically specify an alt value and are ignored by accessible technologies. Graphics that include asterisks or similar as the alt text are slightly better, but are still a suboptimal approach in that they don't convey any meaning except through the reading of the alt value.

There are people who would argue that context breaks represent the borders between untitled subsections within sections, but from a structural and navigational perspective it's typically not true or wanted, so don't be too tempted to add section elements.

HTML5 has, in fact, addressed this need for a transitioning element by changing the semantics of the `hr` element for this purpose:

```
<p>... the world swam and disappeared into darkness.</p>

<hr class="transition"/>

<p class="nonindent">When next we met ...</p>
```

By default this tag would include a horizontal rule, but you can use CSS to turn off the effect and leave a more traditional space for visual viewing:

```
hr.transition {
  width: 0em;
  margin: 0.5em 0em;
}
```

or you could add a fleuron or other ornament:

```
hr.transition {
  background: url('img/fleuron.gif') no-repeat 50% 50%;
  height: 1em;
  margin: 0.5em 0em;
}
```

Styling the `hr` element ensures that the context change isn't lost in the rush to be visually appealing.

Lists

You'd typically not expect to have to hear the advice that you should use lists for sets of related items, but rely too heavily on print tools to create your content and the result will be paragraphs made to look like list items, or single paragraphs that merge all the items together using only `br` tags to separate them.

If you don't use proper list structures, readers can get stuck having to traverse the entire set of items before they can continue with the narrative flow (in the case of one paragraph per item) or having to listen to every item in full to hear the list (when `br` tags are used).

A list element, on the other hand, provides the ability both to move quickly from item to item and to escape the list entirely. It also allows a reading system to inform a reader how many items are in the list and which one they are at for referencing. Picture a list with tens or hundreds of items and you'll get a sense for why this functionality is critical.

Using paragraphs for lists also leads people to resort to visual trickery with margins to emulate the deeper indentation that a nested list would have. These kinds of illusions

take away from all but sighted readers that there exists a hierarchical relationship. The correct tagging allows readers to navigate the various levels with ease.

A final note is to always use the right kind of list:

- the `ol` element is used when the order of the items is important.
- the `ul` element is used when there is no significance or weak significance to the items (e.g., just because you arrange items alphabetically does not impart meaning to the order).
- the `dl` element is used to define terms, mark up glossaries, etc.

Lists have these semantics for good purpose, so don't use CSS to play visual games with them.

Tables

The reflowable, paginated nature of ebook reading has fortunately kept tables from being used for presentational purposes in ebooks. In theory, this should have been a good thing. The complex nature of tables relative to limited rendering area of typical reading systems has led to the worse practice of excluding the data in favor of images of the table, however. How helpful is a picture of data to someone who cannot see it?

The motivating hope behind this practice seems to be that images will take away rendering issues on small screens, but don't fall into this trap. Not only are you taking the content away from readers who can't see the table, but even if you can see the images they often get scaled down to illegibility and/or burst out the side of the reading area on the devices that this technique is presumably meant to enhance the tables on (notably eInk readers that have no zooming functionality).

Consider also what you're doing when you add a picture: you're trying to address a situational disability (the inability to view an entire table at once) by creating another disability (only limited visual access to the content). If you properly mark up your data, readers can find ways to navigate it, whether via synthetic speech or other accessible navigation mechanisms. Obsessing about appearance is natural, but ask yourself how realistic a concern it should be when people read on cellphone screens? Give your readers credit to understand the limitations their devices impose, and give them the flexibility to find other ways to read.

When it comes to marking up tables, the fundamental advice for making them accessible from web iterations past remains true:

- Always use `th` elements for header cells.
- Wrap your header in a `thead`, in particular when including multi-row headings.
- Use the `th scope` attribute to give the applicability of the heading (e.g., whether to the row or column). This attribute is not necessary for simple tables where the first

row of `th` elements, or a `th` cell at the start of each row, defines the header(s), however.

- If the header for a cell cannot be easily determined by its context, and especially when multiple cells in a multi-row header apply, add the `headers` attribute and point to the appropriate `th` element(s).

These heading requirements allow a person navigating your table to quickly determine what they're reading at any given point in it, which is the biggest challenge that tables pose outside of perhaps escaping from them. It's easy to get lost in a sea of numbers, otherwise.

The following example shows how these practices could be applied to a table of baseball statistics:

```
<table>
  <caption>1927 New York Yankees</caption>
  <thead>
    <tr>
      <th rowspan="2">Player</th>
      <th id="reg-hd" colspan="3">Regular Season</th>
      <th id="post-hd" colspan="3">Post Season</th>
    </tr>
    <tr>
      <th id="reg-ab">At Bats</th>
      <th id="reg-hits">Hits</th>
      <th id="reg-avg">Average</th>
      <th id="post-ab">At Bats</th>
      <th id="post-hits">Hits</th>
      <th id="post-avg">Average</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Lou Gehrig</td>
      <td headers="reg-hd reg-ab">584</td>
      <td headers="reg-hd reg-hits">218</td>
      <td headers="reg-hd reg-avg">.373</td>
      <td headers="post-hd post-ab">13</td>
      <td headers="post-hd post-hits">4</td>
      <td headers="post-hd post-avg">.308</td>
    </tr>
  </tbody>
</table>
```

The `headers` attribute on the `td` cells identifies both whether the cell contains a “Regular Season” or “Post Season” statistic as well as the particular kind of stat from the second header row. The value of this tagging is that a reading system or assistive technology can now announce to the reader that they are looking at “regular season hits” when presented the data for the third column, for example.

There's also no reason why this functionality can't be equally useful to sighted readers, except that it's rarely made available. We just talked about the problem of visually

rendering table data on small screens, and there's an obvious solution here to the problem a sighted reader will have of seeing perhaps only a few cells at a time and not having the visual context of what they're looking at. But whether mainstream devices begin taking advantage of this information to solve these problems remains to be seen.

It's also good practice to provide a summary of complex tables to orient readers to their structure and purpose in advance, but the `summary` attribute has been dropped from HTML5. This loss is slightly less objectionable than the `longdesc` attribute removal we'll touch on when we get to images, as prose attributes have many limitations—from expressivity to international language support.

The problem is that HTML5 doesn't replace these removals with any mechanism(s) to allow the discovery of the same information, instead deferring to the `aria-describedby` attribute to point to the information (see the scripting section for more on WAI-ARIA). This attribute, however, may make the information even less generally discoverable to the broader accessibility community, as only persons using accessible technologies will easily find it.

The proposed HTML5 solutions for adding summaries, like using the `caption` element, also don't take into account the need to predictably find this information before presenting the table. The information can't be in any of a number of different places with the onus on the person reading the content to find it.

But throwing our collective hands up in the air isn't a viable solution, either. The `details` element could work as a non-intrusive mechanism for including descriptions, at least until a better solution comes along. This element functions like a declarative show/hide box. Unfortunately, it suffers from a lack of semantic information that the `epub:type` attribute cannot currently remedy (i.e., there are no terms available for identifying whether the element contains a summary or description or something else). We instead have to use a child `summary` element to carry a prose title, as in the following example:

```
<details>
  <summary>Summary</summary>
  <p>...</p>
</details>
```

(The value of the `summary` element represents the clickable text used to expand/close the field and can be whatever you choose.)

If we then take a small liberty with the meaning of the `aria-describedby` attribute to also include summary descriptions, we could reformulate the HTML5 specification example to include an explicit pointer to the `details` element:

```
<table aria-describedby="tbl01-summary">
  <caption>
    Characteristics with positive and negative sides.
  <details id="tbl01-summary">
    <summary>Summary</summary>
    <p>Characteristics are given in the second column...</p>
```

```

        </details>
    </caption>
    ...
</table>

```

In this markup, a nonvisual reader can now find the summary when encountering the table, while a sighted reader will only be presented the option of whether to expand the `details` element. It may not prove a great solution in the long run, but until the landscape settles it's the best on offer.

Figures

Coming up for a quick breath of fresh air before descending into another accessibility attribute pain point, HTML5 introduces the handy new `figure` element for encapsulating content associated with an image, table, or code example. Grouping related content elements together, as is becoming an old theme now, makes it simpler for a reader to navigate and understand your content:

```

<figure>
  
  <figcaption>
    Figure 3.7 &#x2014; The blob is digesting Steve McQueen in this
    unreleased ending to the classic movie.
  </figcaption>
</figure>

```

Unfortunately, there is little support for these two new elements at this time, so they get treated as no better than `div` elements. That said, it's still preferable to future-proof your data and do the right thing, as support will catch up, especially since the only other alternative is semantically meaningless `div` elements.

Images

Images present a challenge for a variety of disabilities, and the means of handling them are not new, but HTML5 has added a new barrier in taking away the `longdesc` attribute for out-of-band descriptions. Like I talked about for tables, you're now left to find ways to incorporate your accessible descriptions in the content of your document.

If only to keep consistent with the earlier suggestion for tables, wrapping the `img` element in a `figure` and using a `details` element as a child of the `figcaption` may suit your needs, as shown in the following example:

```

<figure aria-describedby="fig01-desc">
  
  <figcaption>
    Figure 3.7 – The blob is digesting Steve McQueen in
    this unreleased ending to the classic movie.
    <details id="fig01-desc">
      <summary>Description</summary>
      <p>
        In the photo, Steve McQueen can be seen floating within the

```

```

        gelatinous body of the blob as it moves down the main
        street ...
    </p>
</details>
</figcaption>
</figure>

```

Another option is to include a hyperlinked text label to your long description:

```

<figure>
  <p><a href="blob-desc.xhtml">Description</a></p>
  
  <figcaption>
    Figure 3.7 – The blob is digesting Steve McQueen in this
    unreleased ending to the classic movie.
  </figcaption>
</figure>

```

which would allow the accessible description to live external to the content. You’ll notice I haven’t added an `aria-describedby` attribute to this example because only the prose of the associated element gets presented to a reader using an assistive technology. In this case, the word “Description” would be announced, but the reader would not be presented with the option to link to the description.

Continuing to make the case for `longdesc`, or a better equivalent alternative, is the best course of action, however.

But that muckiness aside, it’s much more pleasant to note that the `alt` attribute has not changed, even if confusion around its use still abounds. The `alt` attribute is not a short description; it’s intended to provide a text equivalent that can replace the image for people for whom the image is not accessible.

Best practices for writing the alternative text extend beyond what we can realistically cover in a guide about EPUB 3, and resources can be easily located on the Web if you’re not clear about the distinction between an alt text and description. A good free reference written by Jukka Korpela is available at <http://www.cs.tut.fi/~jkorpela/html/alt.html>

Of particular note for accessible practices, however, is that even though the `alt` attribute always has to be present on images, it does not always have to contain a text alternative:

```

```

This little fact often gets overlooked. If you add text to an `alt` attribute, you’re indicating that the image is meaningful to the content and requesting that the reader pay attention to it. Images that only exist to make content look pretty should include empty `alt` attributes, as that allows reading systems and assistive technologies to skip readers past them without interrupting their reading experience.

SVG

Rounding out the tour of image functionality is SVG. It comes up for debate every so often just how accessible SVG really is, and while you can argue that it can be more

accessible than non-XML formats like JPEG and PNG, there's no blanket statement like "SVG is completely accessible" that can be applied. Like all content, an SVG is only as accessible as you make it, and when you start scripting one, for example, you can fall into all the typical inaccessibility traps.

The advantages of SVG for accessibility are noteworthy, though. You can scale SVG images without the need for specialized zoom software (and without the typical pixelation effect that occurs when zooming raster formats), the images are accessible technology-friendly when it comes to scripting and can be augmented by WAI-ARIA, and you can add a title and a description directly to the markup without resorting to the messy techniques the `img` element requires:

```
<svg:svg xmlns:svg="http://www.w3.org/2000/svg">
  <svg:title>Figure 1.1, The Hydrologic Cycle</svg:title>
  <svg:desc>
    The diagram shows the processes of evaporation, condensation,
    evapotranspiration, water storage in ice and snow, and
    precipitation. ...
  </svg:desc>
  ...
</svg:svg>
```

Note that the SVG working group also provides a guide to making accessible SVGs that should also be consulted when creating content: <http://www.w3.org/TR/SVG-access/>

The accessibility hooks are also why SVG has been promoted up to a first-class content format (i.e., your ebook can contain only SVG images; they don't have to be embedded in XHTML files). But if you are going to go with an image-only ebook, the quality of your descriptions is going to be paramount, as they will have to tell the story that is lost in your visual imagery. And to be frank, sometimes descriptions will simply fail to capture the richness and complexity of your content, in which case fallback text serializations should be considered.

MathML

Why is MathML important for accessibility? Consider the following simple description of an equation: the square root of a over b . If you hastily added this description to an image of the corresponding equation, what would you expect a reader who couldn't see your image to make of it? Did you mean they should take the square root of a and divide that by b , or did you mean for them to take the square root of the result of dividing a by b ?

The lack of MathML support until now has resulted in these kinds of ambiguities arising in the natural language descriptions that accompanied math images. Ideally your author would describe all their formulas, but the ability to write an equation doesn't always translate into the ability to effectively describe it for someone who can't see it. And sometimes you have to make do with the resources you have available at hand at

the time you generate the ebook, and lacking both academic and description expertise is a recipe for disaster.

MathML takes the ambiguity out of the equation, as assistive technologies have come a long way in terms of being able to voice math equations now. There are even Word plugins that can enable authors to visually create equations for you without having to know MathML, and tools that can convert LaTeX to MathML. The resources are out there to support MathML workflows, in other words.

But although EPUB 3 now provides native support for MathML, it is still a good practice to include an alternate text fallback using the `alttext` attribute, as not all reading systems will support voicing of the markup:

```
<m:math
  xmlns:m="http://www.w3.org/1998/Math/MathML"
  alttext="Frac Root a EndRoot Over b EndFrac">
  <m:mfrac>
    <m:msqrt>
      <m:mtext>a</m:mtext>
    </m:msqrt>
    <m:mi>b</m:mi>
  </m:mfrac>
</m:math>
```



The preceding description was written in MathSpeak. For more information, see the [MathSpeak™ Initiative homepage](#).

If the equation cannot be described within an attribute (e.g., it would surpass the 255 character limit, requires markup elements, like ruby, to fully describe, etc.), it is recommended that the description be written in XHTML and embedded in an `annotation-xml` element as follows:

```
<m:math xmlns:m="http://www.w3.org/1998/Math/MathML">
  <m:semantics>
    <m:mfrac>
      ...
    </m:mfrac>
    <m:annotation-xml
      encoding="application/xhtml+xml"
      name="alternate-representation">
      <span xmlns="http://www.w3.org/1999/xhtml">
        Frac Root a EndRoot Over b EndFrac
      </span>
    </m:annotation-xml>
  </m:semantics>
</m:math>
```

Note that a `semantics` element now surrounds the entire equation. This element is required in order for the addition of the `annotation-xml` element to be valid.

Footnotes

Footnotes present another challenge to reading enjoyment. Prior to EPUB 3, note references could not be easily distinguished from regular hyperlinks, and the notes themselves were typically marked up using paragraphs and divs, which impeded the ability to skip through them or past them entirely.

Picture yourself in a position where you might have to skip a note or two before you can continue reading after every paragraph. And having to manually listen to each new paragraph to determine if it's a note or a continuation of the text. The practice of clumping all notes at the end of a section is slightly more helpful, but still interferes with the content flow however you read.

The `epub:type` attribute helps solve both these problems when used with the new HTML `aside` element, as in the following example:

```
<p>...<a epub:type="noteref" href="#n1">1</a> ...</p>

<aside epub:type="footnote" id="n1">
  ...
</aside>
```

The “noteref” term in the `epub:type` attribute identifies that the link points to a note, which allows a reading system to alert the reader they've encountered a footnote reference. It also provides the reader the ability to tell the reading system to ignore all such links if she wants to read the text through uninterrupted. Don't underestimate the irritation factor of constant note links being announced!

Likewise, the `aside` element has also been identified as a footnote, permitting the reading system to skip it if the reader has chosen to turn off footnote playback. Putting the note in an `aside` also indicates that the content is not part of the main document flow.

But footnotes are often a nuisance for all readers; sighted readers typically care just as little to encounter them in the text. Identifying all your notes could also allow sighted readers to automatically hide them if they prefer them to not be rendered, saving sometimes limited screen space for the narrative prose. A configurable reading system that lets you decide what content you want to see is within reach with semantically meaningful data.

Page Numbering

It might seem odd to talk about page numbering in a digital format guide, but ebooks have been used by students the world over for more than a decade to facilitate their learning in a world only just weaning itself off print. Picture yourself using an ebook in a classroom where print books are still used. When the professor instructs everyone to open their book to a specific page, your ebook will be most unhelpful if you can't find the same location. Or think about trying to quote a passage from a novel in your final

paper and not being able to indicate where in the print source it came from. Page numbers are not an antiquated concept quite yet.

The practice to date has been to include page numbers using anchor tags, as in the following example:

```
<a name="page361"/>
```

But unless a reading system does a heuristic inspection of the `name` attribute's value to see if it starts with "page" or "pg" or "p" there's not going to be a lot of value to this kind of tagging for readers. These kinds of anchor points did give a location for navigating from the NCX page list, and it did keep the number from being rendered, but it's also lost data.

EPUB 3 once again calls on the `epub:type` attribute to include better semantics:

```
<span xml:id="page361" epub:type="pagenumber">361</span>
```

It's now clearly stated what the span contains, and the page number no longer has to be extracted from an attribute and separated from a page identifier label. It's now up to the reader and their reading system to determine when and how to render this information, if at all.

One note when you do include page numbering is to remember that you should also include the ISBN of the source it came from in the package metadata:

```
<dc:source>urn:isbn:9780375704024</dc:source>
```

Inclusion of the ISBN is recommended as it can be used to distinguish between hard-cover and softcover versions, and between different editions, of the source book. All of these typically would have different pagination, which would affect the ability of the reader to accurately synchronize with the print source in use.

This will ensure that students, teachers, professors, and other interested parties can verify whether the digital edition matches the course criteria. Of course, the ideal day coming will be when everyone is using digital editions and sharing bookmarks—and maybe even auto-synchronizing with the professor's edition.

But there are also other settings beyond educational where page number can be useful, too. Reading is also a social activity, and being able to reference by page numbers in leisure books allows for easier participation in reading groups, for example.

The world isn't completely digital yet, so don't dismiss out of hand the need for print-digital referencing when you're producing both formats for a book.

Getting Around: Navigating an EPUB

We've gone over a number of ways to assist in accessible navigation by improving the structure and semantics of your content, but navigating at the content level is only a complement to a good table of contents. EPUB 3 includes a new declarative means of

creating tables of contents called the *navigation document*, which is a fancy way of saying that you can now create a special kind of XHTML document for reading systems to provide their own built-in navigation mechanism(s).



Note that the navigation document is not necessarily the same as the table of contents at the start of the book or at the beginning of a section. The navigation document is primarily intended for reading system use, but can also be included as content if it can serve both roles.

Declarative tables of contents are not new to EPUB 3, however. EPUB 2 had a file format called the NCX for this purpose, which was taken from the DAISY talking book standard (and that format can still be included in EPUB 3 publications for forwards compatibility with older reading systems). But the NCX was a complex solution to a much simpler problem, and actually hindered accessibility in this new context, as its lack of intrinsic display controls led to navigation levels being chopped off to improve visual rendering.

So, to strike back up on a common theme, not all markup is created equal, and the quality of your table of contents for navigation is a reflection whether you put the full structure of your publication in it or not. The new navigation document fortunately gives you the best of both worlds in that it doesn't require the decision to pick either visual clarity or full accessibility to be made.

Let's take a quick tour through the actual creation process to see how this is done for both reading groups.

The navigation document uses the new HTML5 `nav` element to define various kinds of navigation lists, as more than just a table of contents can be included in it. But as a primary table of contents is required in every EPUB, we'll begin by looking at a very minimal example:

```
<nav epub:type="toc">
  <h1>Contents</h1>
  <ol>
    <li>
      <a href="chapter_001.xhtml">Chapter 1. Loomings.</a>
    </li>
    <li>
      <a href="chapter_002.xhtml">Chapter 2. The Carpet-Bag.</a>
    </li>
  </ol>
</nav>
```

The `epub:type` attribute identifies that this `nav` element represents the table of contents (via the “toc” value). But if the rest of the navigation list looks like nothing more than an ordered list of links, that's because that's exactly what navigation lists are. The `nav` element should include a heading, but after that it only ever includes a single `ol` element.

Each list item either contains a single link to a location in the content (as shown in the example above), a link followed by an ordered list of subheadings, or a `span` element (a heading) followed by an ordered list of subheadings. That’s really all there is to building navigation lists.

Let’s take a look at a piece of a more complex table of contents now that we know what we’re looking at:

```
<nav epub:type="toc">
  <h1>The Contents</h1>
  <ol>
    <li>
      <span>SECTION IV FAIRY STORIESMODERN FANTASTIC TALES</span>
      <ol>
        <li>
          <span>Abram S. Isaacs</span>
          <ol>
            <li>
              <a href="section004.xhtml#s190">
                190. A Four-Leaved Clover
              </a>
              <ol>
                <li>
                  <a href="section004.xhtml#s190-1">
                    I. The Rabbi and the Diadem
                  </a>
                </li>
                <li>
                  <a href="section004.xhtml#s190-2">
                    II. Friendship
                  </a>
                </li>
              </ol>
            </li>
          </ol>
        </li>
      </ol>
    </li>
  </ol>
</nav>
```

Here we start with two heading levels, one for the section and another for the author (as indicated by the `span` tags that surround the text). We then have the title of the tale (“A Four-Leaved Clover”), which has additionally been broken down into parts, for a grand total of four levels of navigable content.

But it’s hard enough to format all these lists for the example, let alone display them in a reading device without line wrapping getting in the way. This is the point where aesthetics would win out in the old NCX and the last level would typically be dropped, since it carries the least structurally-important information. But you’d have also just sacrificed completeness for visual clarity, an accessibility no-no. It might not seem like a big issue here, but consider the many levels of depth typical textbooks contain (num-

bered and unnumbered) and how difficult it makes navigating when the structure outline is gone.

The HTML5 `hidden` attribute arrives at this point to save the day. This attribute is the promised solution to indicating where visual display should end without the requirement to remove entries. Since we've decided we only want to visually render down to the level of the tales the author wrote, we can attach the attribute to the ordered list containing the part links. Removing a couple of levels for clarity, our previous example would now be tagged as follows:

```
<li>
  <a href="section004.xhtml#s190">190. A Four-Leaved Clover</a>
  <ol hidden="hidden">
    <li>
      <a href="section004.xhtml#s190-1">I. The Rabbi and the Diadem</a>
    </li>
    <li>
      <a href="section004.xhtml#s190-2">II. Friendship</a>
    </li>
  </ol>
</li>
```

Now all a sighted reader will be presented is the linkable name of the tale (the child ordered lists will be invisible to them), but someone using an assistive technology will still be able to descend to the part level to move around.

Another advantage of this attribute is that it allows you to selectively decide how to hide rendering. For example, if your leveling changes from section to section, you aren't locked into a single "nothing below level 3" approach to tailoring your content. Only the ordered lists you attach the attribute to are hidden from view.

Before turning to the other types of navigation lists you can include in the navigation document, there is one additional accessibility requirement to note. Since the `a` and `span` elements allow all HTML5 inline content, you need to remember not to assume that their content will always be rendered visually. Or, more to the point, to remember that your entries might not voice properly if they include images or MathML or similar.

If the prose you include will present a challenge to voice playback, you need to include a `title` attribute with an equivalent text rendition to use instead:

```
<li><a href="chapter001.xhtml#pi" title="The life of pi">The Life of  $\pi$ </a></li>
```

Some assistive technologies might voice the pi character in this example as the letter "p", for example, which might make sense in a biology book but would be an awkwardly confusing title to announce in a math book.

But to move on from the primary table of contents, there are other ways to help readers navigate your document that are also a benefit to all. The landmarks nav, for example, can provide quick access to key structures. These can be whatever you want, but jumping to the index, glossaries, and other back matter elements are common tasks the

readers may want to perform many times while reading. Think of it like a kind of bookmark list to key structures:

```
<nav epub:type="landmarks">
  <h1>Guide</h1>
  <ol>
    <li>
      <a epub:type="toc" href="contents.xhtml#toc">
        Table of Contents
      </a>
    </li>
    <li>
      <a epub:type="bodymatter" href="chapter001.xhtml#bodymatter">
        Start of Content
      </a>
    </li>
    <li>
      <a epub:type="glossary" href="glossary.xhtml#gloss">
        Glossary
      </a>
    </li>
    <li>
      <a epub:type="index" href="index.xhtml#idx">
        Index
      </a>
    </li>
  </ol>
</nav>
```

You'll notice that, unlike the table of contents example, the `a` tags have `epub:type` attributes attached to them to provide machine-readable semantics. The `epub:type` attribute is required on all links in the landmarks navigation list. The additional semantics are there to help facilitate quick-link options in reading systems so that the landmarks list doesn't have to be opened and navigated manually each time (e.g., a dedicated reading system option to jump to the index).

We touched on the need for page lists in the last section, so I'll only note that a page-list nav should be included if the ebook is part of a dual print-digital workflow:

```
<nav epub:type="page-list">
  <h1>Page List</h1>
  <ol>
    <li><a href="chapter001.xhtml#page001">1</a></li>
    <li><a href="chapter001.xhtml#page002">2</a></li>
    ...
  </ol>
</nav>
```

And don't limit yourself to paper thinking. The navigation document allows any number of useful navigation lists you can devise. Maybe you want to give readers a quick reference to major scenes in your story, for example. There are innumerable ways in which you can expand on this functionality, but semantics and support are going to

take community and player support to implement. But that's true of all new functionality.

The Untold Story: Metadata

Bringing up the topic of metadata usually triggers thoughts about the need to include title and author information in an ebook. While certainly a necessity, this kind of traditional metadata is not what we're going to delve into now.

One of the big issues facing people with disabilities as they try to enter the ebook market is how to discover the quality of the ebooks they want to buy. One ebook is not the same as another, as we've been discussing, and readers need to know what they're getting when they pay for your product. And you should be rewarded for your commitment to accessibility by having your ebooks stand out from the crowd.

Unfortunately, in the past, once you pushed your ebook into a distribution channel, whatever good work you had done to make your content accessible would become indistinguishable from all the inaccessible content out there to the person on the purchasing end. At about the same time that EPUB 3 was being finalized, however, the people at EDItEUR introduced a new set of accessibility metadata for use in ONIX records. This metadata plugs the information gap.



This guide can't possibly explain all of ONIX, nor would it be the appropriate place to do so. If you are not familiar with the standard, please visit the [EDItEUR Web site](#) for more information.

An ONIX record, if you're not familiar, is an industry-standard message (xml record) that accompanies your publication, providing distributors with both content and distribution metadata. This information is typically used to market and sell your ebook, and these new properties now allow you to enhance that information with the accessible features your ebook includes. Retailers can then make this information available in the product description, for example, so readers can determine whether the ebook will adequately meet their needs. It's also not a stretch to imagine this information being integrated into bookstore search engines to allow readers to narrow their results based on their needs.

To include accessibility metadata in your ONIX record, you use the `ProductFormFeature` element, which is described in the standard as a composite container for product information (e.g., it is also used to provide product safety information). To identify that the element contains accessibility information, it must include a `ProductFormFeatureType` child element with the value `09`, as in the following example:

```
<ProductFormFeature>
  <ProductFormFeatureType>09</ProductFormFeatureType>
```

```
...
</ProductFormFeature>
```

When this value is encountered, it indicates that the value of the sibling `ProductFormFeatureValue` element is drawn from [ONIX for Books codelist 196](#), which defines the new accessible properties and the values to use in identifying them. As of writing, this codelist includes the following properties:

- No reading system accessibility options disabled (10)
- Table of contents navigation (11)
- Index navigation (12)
- Reading order (13)
- Short alternative descriptions (14)
- Full alternative descriptions (15)
- Visualised data also available as non-graphical data (16)
- Accessible math content (17)
- Accessible chem content (18)
- Print-equivalent page numbering (19)
- Synchronised pre-recorded audio (20)

So, for example, if our EPUB contains accessible math formulas (using MathML) we would indicate it as follows:

```
<ProductFormFeature>
  <ProductFormFeatureType>09</ProductFormFeatureType>
  <ProductFormFeatureValue>17</ProductFormFeatureValue>
</ProductFormFeature>
```

Likewise, to indicate accessible chemistry content (using ChemML) we would use the value 18:

```
<ProductFormFeature>
  <ProductFormFeatureType>09</ProductFormFeatureType>
  <ProductFormFeatureValue>18</ProductFormFeatureValue>
</ProductFormFeature>
```

You must repeat the `ProductFormFeature` element for each accessibility requirement your EPUB meets; it is *not valid* to combine all your features into a single element.

You must also ensure that you make yourself familiar with the requirements for compliance before adding any accessibility properties; each of the properties defined in codelist 196 includes conformance criteria. EPUB requires table of contents navigation, for example, but the addition of the required navigation document does not automatically indicate compliance. You need to include full navigation, plus navigation to any tables and figures, before you comply with the table of contents navigation requirement. Likewise, all EPUBs have a reading order defined by the spine element in the package document, but that does not mean you comply to including a reading order. You need

to ensure that sidebars and footnotes and similar information has been properly marked up so as not to interfere with the narrative flow, as we went over in the section on the logical reading order.

It is additionally worth noting that there are three properties that aren't directly related to the content of your EPUB:

- Compatibility tested (97)
- Trusted intermediary contact (98)
- Publisher contact for further accessibility information (99)

Each of these properties requires the addition of a child `ProductFormFeatureDescription` element to describe compliance:

```
<ProductFormFeature>
  <ProductFormFeatureType>09</ProductFormFeatureType>
  <ProductFormFeatureValue>97</ProductFormFeatureValue>
  <ProductFormFeatureDescription>
    Content has been tested to work on iBooks, Sony Reader and Adobe Digital
    Editions in both scripting enabled and disabled modes.
  </ProductFormFeatureDescription>
</ProductFormFeature>
```

Whatever compatibility testing you've performed should be listed using code 97 so that readers can better determine the suitability of your ebook for their purposes. Not all reading systems and assistive technologies work alike, or interact well together, and real-world testing is the only way to tell what does and does not work.

There are organizations who specialize in testing across a broad spectrum of devices who can assist you in evaluating your content, as this kind of evaluation can be no small undertaking. A good practice to develop, for example, might be to periodically have typical examples of content coming out of your production stream tested for compatibility issues. The resulting statement could then be re-used across much of your content, so long as the features and markup used remain consistent.

Once you start getting creative with your content (e.g., using extensive scripted interactivity), you should engage experts as early on in the process as you can to ensure any statement you make remains true, however.

The other two codes provide contact information for the person(s) in your organization, or at a trusted intermediary, who can provide additional accessibility information:

```
<ProductFormFeature>
  <ProductFormFeatureType>09</ProductFormFeatureType>
  <ProductFormFeatureValue>99</ProductFormFeatureValue>
  <ProductFormFeatureDescription>
    accessibility-officer@example.com
  </ProductFormFeatureDescription>
</ProductFormFeature>
```

If you're providing educational materials, for example, you'll want to provide educators a way to be able to contact you to determine whether your content will be accessible

by students with various needs that may not be specifically addressed in your compatibility testing statement or in the available metadata properties.

It's Alive: Rich Content Accessibility

It's now time to turn our attention to the features that EPUB 3 introduces to expand on the traditional reading experience; the excitement around EPUB 3 doesn't come from text and images, after all.

This section takes focus on the dynamic aspects of EPUB 3. Rich media, audio integration, and scripted interactivity are all new features that have been added in this version. Some of these features, like audio and video support and scripting, introduce new accessibility challenges, while others, like overlaying audio on your text content and enhancing text-to-speech rendering, improve access for all. (The members of this latter group are also commonly referred to as accessibility superstructures, because they are added on top of core EPUB content to enhance accessibility.)

But let's get back to business of being accessible...

The Sound and the Fury: Audio and Video

The new built-in support for audio and video in EPUB 3 has its pros and cons from both mainstream and accessibility perspectives. The elements simplify the process of embedding multimedia, but come at the expense of complicating interoperability, and by extension accessibility—specifically as relates to video.

There is currently no solution for the general accessibility problem of video, namely that not all reading systems may natively play your content. The video *element* permits any video format to be specified, but not all reading systems will support all formats. Support for one or both of the VP8 codec (used in WebM video) and H.264 codec is encouraged in the EPUB specification to improve interoperability, but you still have to be aware that if you have an EPUB with WebM video and a reading system that only supports H.264 you won't be able to view the video.

Until consensus on codec and container support can be found, there is no easy solution to this problem. You can try targeting your video format to the distribution channel, but that assumes that the readers buying from the online bookstore will use the reading

system you expect, which isn't a given. Even seemingly-simple solutions, like duplicating the format of all video content, are only feasible on small scales and don't take into account the potential cost involved.



It is possible that some reading systems may provide no video support at all.

But that was more of an aside to say that if you don't think accessibility is worth your time and effort, consider there may be a larger audience than you might expect that could be relying on your fallbacks in the near term.

Playability issues aside, though, HTML5 is still a leap forward in terms of multimedia support. It's fair to assert that no one will miss plugins for rendering audio and video content, certainly not on the accessibility side of the fence. From roach-motel players that let you navigate in but never let you leave to players lacking keyboard support to utter black holes, the accessibility community typically does not have a lot of good things to say about multimedia as deployed on the Web.

That the new native elements can be controlled by the reading system in EPUB 3 should translate into greater accessibility, however. To enable the default system controls, you need only add a `controls` attribute to the element:

```
<video ... controls="controls">
```

That these native controls vary in appearance from reading system to reading system, however, leads to a natural tendency to script custom players. There's nothing wrong from an accessibility perspective in doing so, so long as your developers are fluent with WAI-ARIA and ensure the custom controls are fully accessible.

But if you do create a custom control set using JavaScript, ensure that you still enable the native browser controls in the `audio` and `video` elements by default. If you don't, only readers with JavaScript-enabled systems will be able to access the audio or video content, and maybe only some of them. Depending on what scripting capabilities are available, even script-enabled systems may not render your controls. You can always disable the native controls in your JavaScript code if the system supports your custom controls.



See the [epubReadingSystem](#) object for more information on how to query what scripting capabilities a system has.

Timed Tracks

Improved access to the content and the playback controls is only one half of the problem; your content still needs to be accessible to be useful. To this end, both the `audio` and `video` elements allow timed text tracks to be embedded using the HTML5 `track` element.

If you're wondering what timed text tracks are, though, you're probably more familiar with their practical names, like captions, subtitles, and descriptions. A timed track provides the instructions on how to synchronize text (or its rendering) with an audio or video resource: to overlay text as a video plays, to include synthesized voice descriptions, to provide signed descriptions, to allow navigation within the resource, etc.

As I touched on when talking about accessibility at the start of the guide, don't underestimate the usefulness of subtitles and captions. They are not a niche accessibility need. There are many cases where a reader would prefer not to be bothered with the noise while reading, are reading in an environment where it would bother others to enable sound, or are unable to hear clearly or accurately what is going on because of background noise (e.g., on a subway, bus, or airplane). The irritation they will feel at having to return to the video later when they are in a more amenable environment pales next to someone who is not provided any access to that information.

It probably bears repeating at this point, too, that subtitles and captions are not the same thing, and both have important uses that necessitate their inclusion. Subtitles provide the dialogue being spoken, whether in the same language as in the video or translated, and there's typically an assumption the reader is aware which person is speaking. Captions, however, are descriptive and provide ambient and other context useful for someone who can't hear what else might be going on in the video in addition to the dialogue (which typically will shift location on the screen to reflect the person speaking).

A typical aside at this point would be to show a simple example of how to create one of these tracks using one of the many available technologies, but plenty of these kinds of examples abound on the Web. Understanding a bit of the technology is not a bad thing, but, similar to writing effective descriptions for images, the bigger issue is having the experience and knowledge about the target audience to create meaningful and useful captions and descriptions. These issues are outside the realm of EPUB 3, so the only advice I'll give is if you don't have the expertise, engage those who do. Transcription costs are probably much less than you'd expect, especially considering the small amounts of video and audio ebooks will likely include.

We'll instead turn our attention to how these tracks can be attached to your audio or video content using the `track` element. The following example shows a subtitle and caption track being added to a video:

```
<video width="320" height="180" controls="controls">
  <source src="video/v001.webm" type="video/webm; codecs='vp8, vorbis'"/>
  <track
```

```

        kind="subtitles"
        src="video/captions/en/v001.vtt"
        srclang="en"
        label="English"/>
    <track
        kind="captions"
        src="video/captions/en/v001.cc.vtt"
        srclang="en"
        label="English"/>
</video>

```

The first three attributes on the `track` element provide information about the relation to the referenced video resource: the `kind` attribute indicates the nature of the timed track you're attaching; the `src` attribute provides the location of the timed track in the EPUB container; and the `srclang` attribute indicates the language of that track.

The `label` attribute differs in that it provides the text to render when presenting the options the reader can select from. The value, as you might expect, is that you aren't limited to a single version of any one type of track so long as each has a unique label. We could expand our previous example to include translated French subtitles as follows:

```

<video width="320" height="180" controls="controls">
  <source src="video/v001.webm" type="video/webm; codecs='vp8, vorbis'"/>
  <track
    kind="subtitles"
    src="video/captions/en/v001.vtt"
    srclang="en"
    label="English"/>
  <track
    kind="captions"
    src="video/captions/en/v001.cc.vtt"
    srclang="en"
    label="English"/>
  <track
    kind="subtitles"
    src="video/captions/fr/v001.vtt"
    srclang="fr"
    label="Fran&#xE7;ais"/>
</video>

```

I've intentionally only used the language name for the label here to highlight one of the prime deficiencies of the `track` element for accessibility purposes, however. Different disabilities have different needs, and how you caption a video for someone who is deaf is not necessarily how you might caption it for someone with cognitive disabilities, for example.

The weak semantics of the `label` attribute are unfortunately all that is available to convey the target audience. The HTML5 specification, for example, currently includes the following `track` for captions (fixed to be XHTML-compliant):

```

<track
  kind="captions"

```

```
src="brave.en.hoh.vtt"  
srclang="en"  
label="English for the Hard of Hearing"/>
```

You can match the kind of track and language to a reader's preferences, but you can't make finer distinctions about who is the intended audience without reading the label. Machines not only haven't mastered the art of reading, but native speakers find many ways to say the same thing, scuttling heuristic tests.

The result is that reading systems are going to be limited in terms of being able to automatically enable the appropriate captioning for any given user. In reality, getting one caption track would be a huge step forward compared to the Web, but it takes away a tool from those who do target these reader groups and introduces a frustration for the readers in that they have to turn on the proper captioning for each video.

I mentioned the difference between subtitles and captions at the outset, but the `kind` attribute can additionally take the following two values of note:

- `descriptions` — specifying this value indicates that the track contains a text description of the video. A `descriptions` track is designed to provide missing information to readers who can hear the audio but not see the video (which includes blind and low-vision readers, but also anyone for whom the video display is obscured or not available). The track is intended to be voiced by a text-to-speech engine.
- `chapters` — a `chapters` track includes navigational aid within the resource. If your audio or video is structured in a meaningful way (e.g., scenes), adding a `chapters` track will enable readers of all abilities to more easily navigate through it.

But now I'm going to trip you up a bit. The downside of the `track` element that I've been trying to hold off on is that it remains unsupported in browser cores as of writing (at least natively), which means EPUB readers also may not support tracks right away. There are some JavaScript libraries that claim to be able to provide support now (polyfills, as they're colloquially called), but that assumes the reader has a JavaScript-enabled reading system.

Embedding the tracks directly in your video resources is, of course, another option if native support does not materialize right away.

Talk to Me: Media Overlays

When you watch words get highlighted in your reading system as a narrator speaks, the term *media overlay* probably doesn't immediately jump to mind as the best marketing buzzword to describe the experience. But what you are in fact witnessing is a media type (audio) being overlaid on your text content, and tech trumps marketing in standards!

The audio-visual magic that overlays enable in EPUBs is just the tip of the iceberg, however. Overlays represent a bridge between the audio and video worlds, and between mainstream and accessibility needs, which is what really makes this new technology so exciting. They offer accessible audio navigation for blind and low-vision readers. They can improve the reading experience for persons with trouble focusing across a page and improve reading comprehension for many types of readers. They can also provide a unique read-along experience for young readers.

From a mainstream publisher's perspective, overlays provide a bridge between audio-book and ebook production streams. Create a single source using overlays and you could transform and distribute across the spectrum from audio-only to text-only. With full-text and full-audio synchronization ebooks, you can transform down to a variety of formats. If you're going to create an audiobook version of your ebook, it doesn't make sense not to combine production, which is exactly what EPUB 3 allows you to now do. Your source content is more valuable by virtue of its completeness, and you can also choose to target and distribute your ebook with different modalities enabled for different audiences.

From a reader's perspective, they can purchase a format that provides adaptive modalities to meet their reading needs: they can choose which playback format they prefer, or purchase a book with multiple modalities and switch between them as the situation warrants—listening while driving and visually reading when back at home, for example.

Media overlays are the answer to a lot of problems on both sides of the accessibility fence, in other words.

If you're coming to this guide from accessibility circles, however, you're probably wondering why this is considered new and exciting when it sounds an awful lot like the SMIL technology that has been at the core of the DAISY talking book specifications for more than a decade. And you're right...sort of. Overlays are not new technology, but represent a new evolution of the DAISY standard, which EPUB 3 is a successor to. What is really exciting from an accessibility perspective is the chance to move this production back to the source to get high-quality audio and text synchronized ebooks directly from publishers. Removing synchronization information from having to be encoded in content files is another benefit overlays provide over older talking book formats, greatly simplifying their creation.

But knowing what overlays are and how they can enhance ebooks doesn't get us any closer to understanding how they work and the considerations involved in making them, which is the real goal for this section. If you like to believe in magic, though, here's an early warning that by the end it won't seem all that fantastic how your reading system makes words and paragraphs highlight as a voice narrates the text. Prepare to be disappointed that your reading system doesn't have superpowers.

To begin moving under the hood of an EPUB, though, the first thing to understand is that overlays are just specialized xml documents that contain the instructions a reading

system uses to synchronize the text display with the audio playback. They're expressed using a subset of SMIL that we'll cover as we move along, combined with the `epub:type` attribute we ran into earlier for semantic inflection.



SMIL (pronounced “smile”) is the shorthand way of referring to the Synchronized Multimedia Integration Language. For more information on this technology, see <http://www.w3.org/TR/SMIL>

The order of the instructions in the overlay document defines the reading order for the ebook when in playback mode. A reading system will move through the instructions one at a time, or a reader can manually navigate in similar fashion to how assistive technologies enable navigation through the markup (i.e., escaping and skipping).

As a reading system encounters each synchronization point, it determines from the provided information which element in which content file has to be loaded (by its `id`) and the corresponding position in the audio track at which to start the narration. The reading system will then load and highlight the word or passage for you at the same time that you hear the audio start. When the audio track reaches the end point you've specified—or the end of the audio file if you haven't specified one—the reading system checks the next synchronization point to see what text and audio to load next.

This process of playback and resynchronization continues over and over until you reach the end of the book, giving the appearance to the reader that their system has come alive and is guiding them through it.



This portrayal is intentionally simple. In practice, overlay synchronization points may, for example, omit an audio reference when the reading system is expected to synthetically render the text, or if the text reference points to a multimedia object like the audio or `video` element that the reading system is expected to initiate. Refer to the Media Overlays specification for more information on the full range of features.

As you might suspect at this point, the reading system can't synchronize or play content back any way but what has been defined; as a reader you cannot, for example, dynamically change from word-to-word to paragraph-by-paragraph read-back as you desire. The magic is only as magical as you make it, at least at this time.

With only a single level of playback granularity available, the decision on how fine a playback experience to provide has typically been influenced by the disability you're targeting, going back to the origins of the functionality in talking books. Books for blind and low-vision readers are often only synchronized to the heading level, for example, and omit the text entirely. Readers with dyslexia or cognitive issues, however, may benefit more from word-level synchronization using full-text full-audio playback.

Coarser synchronization—for example at the phrase or paragraph level—can be useful in cases where the defining characteristics of a particular human narration (flow, intonation, emphasis) add an extra dimension to the prose, such as with spoken poetry or religious verses. The production costs associated with synchronizing human-narrated ebooks to the word level, however, has typically meant that only short-prose works (such as children’s books) get this treatment.

Let’s turn to the practical construction of an overlay to discover why the complexity increases by level, though. Understanding the issues will give better insight into which model you ultimately decide to use.

Building an Overlay

Every overlay document begins with root `smil` element and a `body`, as exemplified in the following markup:

```
<smil
  xmlns="http://www.w3.org/ns/SMIL"
  xmlns:epub="http://www.idpf.org/2007/ops"
  version="3.0">
  <body>

  </body>
</smil>
```

There’s nothing exciting going on here but a couple of namespace declarations and a `version` attribute on the root. These are static in EPUB 3, so of little interest beyond their existence. There is no required metadata in the overlays themselves, which is why we don’t need to add a `head` element.

Of course, in order to now illustrate how to build up this shell and include it in an EPUB, we’re going to need some content. I’m going to use the *Moby Dick* ebook that Dave Cramer, a member of the EPUB working group, built as a proof of concept of the specification for the rest of this section. This book is available from the [EPUB 3 Sample Content project page](#).

If we look at the content file for chapter one, we can see that the HTML markup has been structured to showcase different levels of text/audio synchronization. After the chapter heading, for example, the first paragraph has been broken down to achieve fine synchronization granularity (word and sentence level), whereas the following paragraph hasn’t been divided into smaller parts.

Compressing the markup in the file to just what we’ll be looking at, we have:

```
<section id="c01">
  <h1 id="c01h01">Chapter 1. Loomings.</h1>

  <p><span id="c01w00001">Call</span>
    <span id="c01w00002">me</span>
    <span id="c01w00003">Ishmael.</span>
    <span id="c01s0002">Some years ago...</span> ...</p>
```

```
<p id="c01p0002">There now is your insular city of the Manhattoes...</p>
...
</section>
```

You'll notice that each element containing text content has an `id` attribute, as that's what we'll be referencing when we get to synchronizing with the audio track.

The markup additionally includes `span` tags to differentiate words and sentences in the first `p` tag. The second paragraph only has an `id` attribute on it, however, as we're going to omit synchronization on the individual text components it contains to show paragraph-level synchronization.

We can now take this information to start building the body of our overlay. Switching back to our empty overlay document, the first element we're going to include in the body is a `seq`:

```
<body>
  <seq
    id="id1"
    epub:textref="chapter_001.xhtml#c01"
    epub:type="bodymatter chapter">
  </seq>
</body>
```

This element serves the same grouping function the corresponding `section` element does in the markup, and you'll notice the `textref` attribute references the `section`'s `id`. The logical grouping of content inside the `seq` element likewise enables escaping and skipping of structures during playback, as we'll return to when we look at some structural considerations later.

In this case, the `epub:type` attribute conveys that this `seq` represents a chapter in the body matter. Although the attribute isn't required, there's little benefit in adding `seq` elements if you omit any semantics, as a reading system will not be able to provide skippability and escapability behaviors unless it can identify the purpose of the structure.

It may seem redundant to have the same semantic information in both the markup and overlay, but remember that each is tailored to different rendering and playback methods. Without this information in the overlay, the reading system would have to inspect the markup file to determine what the synchronization elements represent, and then resynchronize the overlay using the markup as a guide. Not a simple process. A single channel of information is much more efficient, although it does translate into a bit of redundancy (you also typically wouldn't be crafting these documents by hand, and a recording application could potentially pick up the semantics from the markup and apply them to the overlay for you).

We can now start defining synchronization points by adding `par` elements to the `seq`, which is the only other step in the process. Each `par` contains a child `text` and a child

`audio` element, which define the fragment of your content and the associated portion of an audio file to render in parallel, respectively.

Here's the entry for our primary chapter heading, for example:

```
<par id="heading1">
  <text src="chapter_001.xhtml#c01h01"/>
  <audio
    src="audio/mobydick_001_002_melville.mp4"
    clipBegin="0:00:24.500"
    clipEnd="0:00:29.268"/>
</par>
```

The `text` element contains an `src` attribute that identifies the filename of the content document to synchronize with, and a fragment identifier (the value after the `#` character) that indicates the unique identifier of a particular element within that content document. In this case, we're indicating that `chapter_001.xhtml` needs to be loaded and the element with the id `c01h01` displayed (the `h1` in our sample content, as expected).

The `audio` element likewise identifies the source file containing the audio narration in its `src` attribute, and defines the starting and ending offsets within it using the `clipBegin` and `clipEnd` attributes. As indicated by these attributes, the narration of the heading text begins at the mid 24 second mark (to skip past the preliminary announcements) and ends just after the 29 second mark. The milliseconds on the end of the start and end values give an idea of the level of precision needed to create overlays, and why people typically don't mark them up by hand. If you are only as precise as a second, the reading system can move readers to the new prose at the wrong time or start narration in the middle of a word or at the wrong word.

But those concerns aside, that's all there is to basic text and audio synchronization. So, as you can now see, no reading system witchcraft was required to synchronize the text document with its audio track! Instead, the audio playback is controlled by timestamps that precisely determine how an audio recording is mapped to the text structure. Whether synchronizing down to the word or moving through by paragraph, this process doesn't change.

To synchronize the first three words "Call me Ishmael" in the first paragraph, for example, we simply repeat the process of matching element ids and audio offsets:

```
<par>
  <text src="chapter_001.xhtml#c01w00001"/>
  <audio
    src="audio/mobydick_001_002_melville.mp4"
    clipBegin="0:00:29.269"
    clipEnd="0:00:29.441"/>
</par>
<par>
  <text src="chapter_001.xhtml#c01w00002"/>
  <audio
    src="audio/mobydick_001_002_melville.mp4"
    clipBegin="0:00:29.441"
    clipEnd="0:00:29.640"/>
```

```

</par>
<par>
  <text src="chapter_001.xhtml#c01w00003"/>
  <audio
    src="audio/mobydick_001_002_melville.mp4"
    clipBegin="0:00:29.640"
    clipEnd="0:00:30.397"/>
</par>

```

You'll notice each `clipEnd` matches the next element's `clipBegin` here because we have a single continuous playback track. Finding each of these synchronization points manually is not so easy, though, as you might imagine.

Synchronizing to the sentence level, however, means only one synchronization point is required for all the words the sentence contains, thereby reducing the time and complexity of the process several magnitudes. The `par` is otherwise constructed exactly like the previous example:

```

<par>
  <text src="chapter_001.xhtml#c01s0002"/>
  <audio
    src="audio/mobydick_001_002_melville.mp4"
    clipBegin="0:00:30.397"
    clipEnd="0:00:44.783"/>
</par>

```

The process of creating overlays is only complicated by the time and text synchronizations involved, as is no doubt becoming clearer. Moving up another level, paragraph level synchronization reduces the process several more magnitudes as all the sentences can be skipped. Here's the single entry we'd only have to make for the entire 28s second paragraph:

```

<par>
  <text src="chapter_001.xhtml#c01p0002"/>
  <audio
    src="audio/mobydick_001_002_melville.mp4"
    clipBegin="0:01:46.450"
    clipEnd="0:02:14.138"/>
</par>

```

The complexity isn't only limited to the number of entries and finding the audio points, however, otherwise technology would easily overcome the problem. Narrating at a heading, paragraph, or even sentence level can be done relatively easily with trained narrators, as each of these structures provides a natural pause point for the person reading, a simplifier not provided when performing word-level synchronization.

A real-world recording scenario, for example, would typically involve the narrator loading their ebook and synchronizing the text in the recording application as they narrate to speed up this process immensely (e.g., using the forward arrow or spacebar each time they start a new paragraph to have the recording program automatically set the new synchronization point). Performing the synchronization at the natural pause points is not problematic in this scenario, as the person reading is briefly not focused

on that task and/or the person assisting has enough of a break to cleanly resynchronize. Trying to narrate and synchronize at the word level, however, is a tricky process to perform effectively, as people naturally talk more fluidly than any process can keep up with, even if two people are involved.



The real-world experience I describe here comes from the creation of DAISY talking books, to be clear. Tools for the similar production of EPUB 3 overlays will undoubtedly appear in time, as well, but as of writing are in short supply.

Ultimately, the only advice that can be given is to strive for the finest granularity you can. Paragraphs may be easier to synchronize than sentences, but if the viewing screen isn't large enough to view the entire paragraph the invisible part won't ever come into view as the narration plays (the reading system can only know to resynch at the next point; it can't intrinsically know that narration has to match what is on screen or have any way to determine what is on screen at any given time).

We're not completely done yet, though. There are a few quick details to run through in order to now include this overlay in our EPUB.



The following instructions assume a basic level of familiarity with EPUB publication files. Refer to the [EPUB Publications 3.0 specification](#) for more information.

Assuming we've saved our overlay as *chapter_001_overlay.smil*, the first step is simply to include an entry in the manifest:

```
<item
  id="chapter_001_overlay"
  href="chapter_001_overlay.smil"
  media-type="application/smil+xml"/>
```

We then need to include a `media-overlay` attribute on the manifest item for the corresponding content document for chapter one:

```
<item
  id="xchapter_001"
  href="chapter_001.xhtml"
  media-type="application/xhtml+xml"
  media-overlay="chapter_001_overlay"/>
```

The value of this attribute is the id we created for the overlay in the previous step.

And finally we need to add metadata to the publication file indicating the total length of the audio for each individual overlay and for the publication as a whole. For completeness, we'll also include the name of the narrator.

```

<meta property="media:duration" refines="#chapter_001_overlay">0:14:43</meta>
...
<meta property="media:duration">0:23:46</meta>
<meta property="media:narrator">Stuart Wills</meta>

```

The `refines` attribute on the first `meta` element specifies the id of the manifest item we created for the overlay, as this is how we match the time value up to the content file it belongs to. The lack of a `refines` attribute on the next duration `meta` element indicates it contains the total time for the publication (only one can omit the `refines` attribute).

There's one final metadata item left to add and then we're done:

```

<meta property="media:active-class">-epub-media-overlay-active</meta>

```

This special `media:active-class` meta property tells the reading system which CSS class to apply to the active element when the audio narration is being played (i.e., the highlighting to give it).

For example, to apply a yellow background to each section of prose as it is read, as is traditionally found in accessible talking books, you would define the following definition in your CSS file:

```

.-epub-media-overlay-active
{
    background-color: yellow;
}

```

And that's the long and short of creating overlays.

Structural Considerations

I briefly touched on the need to escape nested structures, and skip unwanted ones, but let's go back to this functionality as a first best practice, as it is critical to the usability of the overlays feature in exactly the same way markup is to content-level navigation.

If you have the bad idea in your head that only `par` elements matter for playback, and you can go ahead and make overlays that are nothing more than a continuous sequence of these elements, get that idea back out of your head. It's the equivalent of tagging everything in the body of a content file using `div` or `p` tags.

Using `seq` elements for problematic structures like lists and tables provides the information a reading system needs to escape from them.

Here's how to structure a simple list, for example:

```

<seq id="seq002" epub:type="list" epub:textref="chapter_012.xhtml#0101">
  <par id="list001item001" epub:type="list-item">
    <text src="chapter_012.xhtml#0101i01"/>
    <audio src="audio/chapter12.mp4" clipBegin="0:26:48" clipEnd="0:27:11"/>
  </par>
  <par id="list001item002" epub:type="list-item">
    <text src="chapter_012.xhtml#0101i02"/>
    <audio src="audio/chapter12.mp4" clipBegin="0:27:11" clipEnd="0:27:29"/>
  </par>
</seq>

```

```
    </par>
    ...
  </seq>
```

A reading system can now discover from the `epub:type` attribute the nature of the `seq` element and of each `par` it contains. If the reader indicates at any point during the playback of the `par` element list items that they want to jump to the end, the reading system simply continues playback at the next `seq` or `par` element following the parent `seq`. If the list contained sub-lists, you could similarly wrap each in its own `seq` element to allow the reader to escape back up through all the levels.

A similar nested `seq` process is critical for table navigation: a `seq` to contain the entire table, individual `seq` elements for each row, and table-cell semantics on the `par` elements containing the text data.

A simple three-cell table could be marked up using this model as follows:

```
<seq epub:type="table" epub:textref="ch007.xhtml#tbl01">
  <seq epub:type="table-row" epub:textref="ch007.xhtml#tbl01r01">
    <par epub:type="table-cell">...</par>
    <par epub:type="table-cell">...</par>
    <par epub:type="table-cell">...</par>
  </seq>
</seq>
```

You could also use a `seq` for the table cells if they contained complex data:

```
<seq epub:type="table-cell" epub:textref="ch007.xhtml#tbl01r01c01">
  <par>...</par>
  ...
</seq>
```

But attention shouldn't only be given to `seq` elements when it comes to applying semantics. Readers also benefit when `par` elements are identifiable, particularly for skipping:

```
<par id="note21" epub:type="note">
  <text src="notes.xhtml#c02note03"/>
  <audio src="audio/notes.mp4" clipBegin="0:14:23.146" clipEnd="0:15:11.744"/>
</par>
```

If all notes receive the semantic as in the above example, a reader could disable all note playback, ensuring the logical reading order is maintained. All secondary content that isn't part of the logical reading order should be so identified so that it can be skipped.

This small extra effort to mark up structures and secondary content does a lot to make your content more accessible.

Tell It Like It Is: Text-to-Speech (TTS)

An alternative (and complement) to human narration, and the associated costs of creating and distributing it, is speech synthesis—when done right, that is. The mere

thought of synthesized speech is enough to make some people cringe, though, as it's still typically equated with the likes of poor old much-maligned Microsoft Sam and his tinny, often-incomprehensible renderings. Modern high-end voices are getting harder and harder to distinguish as synthesized, however, and the voices on most reading systems and computers are getting progressively more natural sounding and pleasant to the ears for extended listening.

But whatever you think of the voices, the need to be able to synthesize the text of your ebook is always going to be vital to a segment of your readers, especially when human narration is not available. It's also generally useful to the broader reading demographic, as I'll return to.

And the voice issues are a bit of a red herring. The real issue here is not how the voices sound but the mispronunciations the rendering engines make, and the frequency with which they often make them. The constant mispronunciation of words disrupts comprehension and ruins reading enjoyment, as it breaks the narrative flow and leaves the reader to guess what the engine was actually trying to speak. It doesn't have to be this way, though; the errors occur because the mechanisms to enhance default synthetic renderings haven't been made available in ebooks, not because there aren't any.

But to step back slightly, synthetic speech engines aren't inherently riddled with errors, they just fail because word pronunciation can be an incredibly complex task, one that requires more than just the simple recognition of character data. Picture yourself learning a new language and struggling to understand why some vowels are silent in some situations and not in others, or why their pronunciation changes in seemingly haphazard ways, not to mention trying to grasp where phonetic boundaries are and so on. A rendering engine faces the same issues with less intelligence and no ability to learn on its own or from past mistakes.

The issue is often sometimes as simple as not being able to parse parts of speech. For example, consider the following sentence:

An official group *record* of past achievements was never kept.

A speech engine may or may not say "record" properly, because record used as noun is not pronounced the same way as when used as a verb in English.

The result is that most reading systems with built-in synthetic speech capabilities will do a decent job with the most common words in any language, but can trip over themselves when trying to pronounce complex compound words, technical terms, proper names, abbreviations, numbers, and the like. Heteronyms—words that are spelled the same way but have different pronunciations and meanings—also offer a challenge, as you can't always be sure which pronunciation will come out. The word *bass* in English, for example, is pronounced one way to indicate a fish (bass) and another to indicate an instrument (base).

When you add up the various problem areas, it's not a surprise why there's a high frequency of errors. These failings are especially problematic in educational, science,

medical, legal, tax, and similar technical publishing fields, as you might expect, as the proper pronunciation of terms is critical to comprehension and being able to communicate with peers.

The ability to correctly voice individual words is a huge benefit to all readers, in other words, which is why you should care about the synthetic rendering quality of your ebooks, as I said I'd get back to. Even if all your readers aren't going to read your whole book via synthetic speech, everyone comes across words they aren't sure how to pronounce, weird-looking character names, etc. In the print world, they'd just have to guess at the pronunciation and live with the nuisance of wondering for the rest of the book whether they have it right in their head or not (barring the rare pronunciation guide in the back, of course).

The embedded dictionaries and pronunciations that reading systems offer are a step up from print, but typically are of little-to-no help in many of these cases, since specialized terms and names don't appear in general dictionaries. Enhancing your ebooks even just to cover the most complicated names and terms goes a long way to making the entire experience better for all. Enhanced synthetic speech capabilities are a great value-add to set you apart from the crowd, especially if you're targeting broad audience groups.

Synthetic speech can also reduce the cost to produce audio-enhanced ebooks. Human narration is costly, as I mentioned at the outset, and typically only practical for novels, general non-fiction, and the like. But even in those kinds of books, are you going to have a person narrate the bibliographies and indexes and other complex structures in the back matter, or would it make more sense to leave them to the reader's device to voice? Having the pronunciation of words consistent across the human-machine divide takes on a little more importance in this light, unless you want to irk your readers with rotten sounding back matter (or worse, omitted material).

And as I mentioned in the overlays section, there are reading systems that already give word-level text-audio synchronization in synthetic speech playback mode, surpassing what most people would attempt with an overlay and human narration. As each word is fed for rendering it gets highlighted on the screen auto-magically; there's nothing special you have to do.

The cost and effort to improve synthetic speech is also one that has the potential to decrease over time as you build re-usable lexicons and processes to enhance your books.

But enough selling of benefits. You undoubtedly want to know how EPUB 3 helps you, so let's get on with the task.

The new specification adds three mechanisms specifically aimed at synthetic speech production: PLS lexicon files, SSML markup, and CSS3 Speech style sheets. We'll go into each of these in turn and explore how you can now combine them to optimize the quality of your ebooks.

PLS Lexicons

The first of the new synthetic speech enhancement layers we'll look at is PLS files, which are xml lexicon files that conform to the [W3C Pronunciation Lexicon Specification](#). The entries in these files identify the word(s) to apply each pronunciation rule to. The entries also include the correct phonetic spelling, which provides the text-to-speech engine with the proper pronunciation to render.

Perhaps a simpler way of thinking about PLS files, though, is as containing globally-applicable pronunciation rules: the entries you define in these files will be used for all matching cases in your content. Instead of having to add the pronunciation over and over every time the word is encountered in your markup, as SSML requires, these lexicons are used as global lookups.

PLS files are consequently the ideal place to define all the proper names and technical terms and other complex words that do not change based on the context in which they are used. Even in the case of heteronyms, it's good to define the pronunciation you deem the most commonly used in your PLS file, as it may be the only case in your ebook(s). It also ensures that you know how the heteronym will always be pronounced by default, to remove the element of chance.



The PLS specification does define a role attribute to enable context-dependent pronunciations (e.g., to differentiate the pronunciation of a word when used as a verb or noun), but support for it is not widespread and no vocabulary is defined for standard use. I'll defer context-dependent differentiation to SSML, as a result, even though a measure is technically possible in PLS files.

But let's take a look at a minimal example of a complete PLS file to see how they work in practice. Here we'll define a single entry for "acetaminophen" to cure our pronunciation headaches:

```
<lexicon
  version="1.0"
  alphabet="x-sampa"
  xml:lang="en"
  xmlns="http://www.w3.org/2005/01/pronunciation-lexicon">
  <lexeme>
    <grapheme>acetaminophen</grapheme>
    <phoneme>@"sit@mIn@f@n</phoneme>
  </lexeme>
</lexicon>
```

To start breaking this markup down, the `alphabet` attribute on the root `lexicon` element defines the phonetic alphabet we're going to use to write our pronunciations. In this case, I'm indicating that I'm going to write them using X-SAMPA.



X-SAMPA is the Extended Speech Assessment Methods Phonetic Alphabet. Being an ASCII-based phonetic alphabet, I've chosen to use it here only because it is more easily writable (by mere mortals like this author) than the International Phonetic Alphabet (IPA). It is not clear at this time which alphabet(s) will receive the most widespread support in reading systems, however.

The `version` and `xmlns` namespace declaration attributes are static values, so nothing exciting to see there, as usual. The `xml:lang` attribute, however, is required, and must reflect the language of the entries contained in the lexicon. Here we're declaring that all the entries are in English.

The root would normally contain many more `lexeme` elements than in this example, as each defines the word(s) the rule applies to in the child `grapheme` element(s). (Graphemes, of course, don't have to take the form of words, but for simplicity of explanation I'll stick to the general concept.) When the string is matched, the pronunciation in the `phoneme` element gets rendered in place of the default rendering the engine would have performed.

Or, if it helps conceptualize, when the word "acetaminophen" is encountered in the prose, before passing the word to the rendering engine to voice, an internal lookup of the defined graphemes occurs. Because we've defined a match, the phoneme and the alphabet it adheres to are swapped in instead for voicing.

That you can include multiple graphemes may not seem immediately useful, but it enables you to create a single entry for regional variations in spelling, for example. British and American variants of "defense" could be defined in a single rule as:

```
<lexeme>
  <grapheme>defense</grapheme>
  <grapheme>defence</grapheme>
  <phoneme>dI'fEns</phoneme>
</lexeme>
```

It is similarly possible to define more than one pronunciation by adding multiple `phoneme` elements. We could add the IPA spelling to the last example as follows, in case reading systems end up only supporting one or the other alphabet:

```
<lexeme>
  <grapheme>defense</grapheme>
  <grapheme>defence</grapheme>
  <phoneme>dI'fEns</phoneme>
  <phoneme alphabet="ipa">di'fɛns</phoneme>
</lexeme>
```

The `alphabet` attribute on the new `phoneme` element is required because its spelling doesn't conform to the default defined on the root. If the rendering engine doesn't support X-SAMPA, it could now possibly make use of this embedded IPA version instead.

The phoneme doesn't have to be in another alphabet, however; you could add a regional dialect as a secondary pronunciation, for example. The specification unfortunately doesn't provide any mechanisms to indicate why you've included such additional pronunciations or when they should be used, so there's not much value in doing so at this time.

There's much more to creating PLS files than can be covered here, of course, but you're now versed in the basics and ready to start compiling your own lexicons. You only need to attach your PLS file to your publication to complete the process of enhancing your ebook.

The first step is to include an entry for the PLS file in the EPUB manifest:

```
<item href="EPUB/lexicon.pls" id="pls" media-type="application/pls+xml"/>
```

The `href` attribute defines the location of the file relative to the EPUB container root and the `media-type` attribute value "application/pls+xml" identifies to a reading system that we've attached a PLS file.

Including one or more PLS files does not mean they apply by default to all your content, however; in fact, they apply to none of it by default. You next have to explicitly tie each PLS lexicon to each XHTML content document it is to be used with by adding a `link` element to the document's header:

```
<html ...>
  <head>
    ...
    <link
      rel="pronunciation"
      href="lexicon.pls"
      type="application/pls+xml"
      hreflang="en" />
    ...
  </head>
  ...
</html>
```

There are a number of differences between the declaration for the PLS file in the publication manifest above and in the content file here. The first is the use of the `rel` attribute to include an explicit relationship (that the referenced file represents pronunciation information). This attribute represents somewhat redundant information, however, since the media type is once again specified (here in the `type` attribute). But as it is a required attribute in HTML5, it can't be omitted.

You may have also noticed that the location of the PLS file appears to have changed. We've dropped the EPUB subdirectory from the path in the `href` attribute because reading systems process the EPUB container differently than they do content files. Resources listed in the manifest are referenced by their location from the container root. Content documents, on the other hand, reference resources relative to their own location in the container. Since we'll store both our content document and lexicon file in the EPUB subdirectory, the `href` attribute contains only the filename of the PLS lexicon.

The HTML link element also includes an additional piece of information to allow selective targeting of lexicons: the `hreflang` attribute. This attribute specifies the language to which the included pronunciations apply. For example, if you have an English document (as defined in the `xml:lang` attribute on the `html` root element) that embeds French prose, you could include two lexicon files:

```
<link
  rel="pronunciation"
  href="lexicon-en.pls"
  type="application/pls+xml"
  hreflang="en" />
```

```
<link
  rel="pronunciation"
  href="lexicon-fr.pls"
  type="application/pls+xml"
  hreflang="fr" />
```

Assuming all your French passages have `xml:lang` attributes on them, the reading system can selectively apply the lexicons to prevent any possible pronunciation confusion:

```
<p>It's the Hunchback of <i xml:lang="fr">Notre Dame</i> not of Notre Dame.</p>
```

A unilingual person reading this prose probably would not understand the distinction being made here: that the French pronunciation is not the same as the Americanization. Including separate lexicons by language, however, would ensure that readers would hear the Indiana university name differently than the French cathedral if they turn on TTS:

```
<lexicon
  version="1.0"
  alphabet="x-sampa"
  xml:lang="en"
  xmlns="http://www.w3.org/2005/01/pronunciation-lexicon">
  <lexeme>
    <grapheme>Notre Dame</grapheme>
    <phoneme>noUt@r 'deIm</phoneme>
  </lexeme>
</lexicon>
```

```
<lexicon
  version="1.0"
  alphabet="x-sampa"
  xml:lang="fr"
  xmlns="http://www.w3.org/2005/01/pronunciation-lexicon">
  <lexeme>
    <grapheme>Notre Dame</grapheme>
    <phoneme>n%oUtr@ d"Am</phoneme>
  </lexeme>
</lexicon>
```

When the contents of the `i` tag are encountered, and identified as French, the pronunciation from the corresponding lexicon gets applied instead of the one from the default English lexicon.

Now that we know how to globally define pronunciation rules, let's turn to how we can override and/or define behavior at the markup level.

SSML

Although PLS files are a great way to globally set the pronunciation of words, their primary failing is that they aren't a lot of help where context matters in determining the correct pronunciation. Leave the pronunciation of heteronyms to chance, for example, and you're invariably going to be disappointed by the result; the cases where context might not significantly influence comprehension (e.g., an English heteronym like "mobile"), are going to be dwarfed by the ones where it does.

By way of example, when talking about PLS files I mentioned bass the instrument and bass the fish as an example of how context influences pronunciation. Let's take a look at this problem in practice now:

```
<p>The guitarist was playing a bass that was shaped like a bass.</p>
```

Human readers won't have much of a struggle with this sentence, despite the contrived oddity of it. A guitarist is not going to be playing a fish shaped like a guitar, and it would be strange to note that the bass guitar is shaped like a bass guitar. From context you're able to determine without much pause that we're talking about someone playing a guitar shaped like a fish.

All good and simple. Now consider your reaction if, when listening to a synthetic speech engine pronounce the sentence, you heard both words pronounced the same way, which is the typical result. The process to correct the mistake takes you out of the flow of the narrative. You're going to wonder why the guitar is shaped like a guitar, admit it.

Synthetic narration doesn't afford you the same ease to move forward and back through the prose that visual reading does, as words are only announced as they're voiced. The engine may be applying heuristic tests to attempt to better interpret the text for you behind the scenes, but you're at its mercy. You can back up and listen to the word again to verify whether the engine said what you thought it did, but it's an intrusive process that requires you to interact with the reading system. If you still can't make sense of the word, you can have the reading system spell it out as a last resort, but now you're train of thought is completely on understanding the word.

And this is an easy example. A blind reader used to synthetic speech engines would probably just keep listening past this sentence having made a quick assumption that the engine should have said something else, for example, but that's not a justification for neglect. The problems only get more complex and less avoidable, no matter your familiarity. And that you're asking your readers to compensate is a major red flag you're not being accessible, as mispronunciations are not always easily overcome depending on the reader's disability. It also doesn't reflect well on your ebooks if readers turn to synthetic speech engines to help with pronunciation and find gibberish, as I touched on in the last section.

And the problems are rarely one-time occurrences. When the reader figures out what the engine was trying to say they will, in all likelihood, have to make a mental note on how to translate the synthetic gunk each time it is re-encountered to avoid repeatedly going through the same process. If you don't think that makes reading comprehension a headache, try it sometime.

But this is where the Synthetic Speech Markup Language (SSML) comes in, allowing you to define individual pronunciations at the markup level. EPUB 3 adds the `ssml:alphabet` and `ssml:ph` attributes, which allow you to specify the alphabet you're using and phonemic pronunciation of the containing element's content, respectively. These attributes work in very much the same way as the PLS entries we just reviewed, as you might already suspect.

For example, we could revise our earlier example as follows to ensure the proper pronunciation for each use of bass:

```
<p>
  The guitarist was playing a
  <span ssml:alphabet="x-sampa" ssml:ph="beIs">bass</span> that was shaped
  like a <span ssml:alphabet="x-sampa" ssml:ph="b&amp;s">bass</span>.
</p>
```

The `ssml:alphabet` attribute on each `span` element identifies that the pronunciation carried in the `ssml:ph` attribute is written in X-SAMPA, identically to the PLS `alphabet` attribute. We don't need a grapheme to match against, because we're telling the synthetic speech engine to replace the content of the `span` element. The engine will now voice the provided pronunciations instead of applying its own rules. In other words, no more ambiguity and no more rendering problem; it really is that simple.



The second `ssml:ph` attribute includes an `&` entity as the actual X-SAMPA spelling is: `b&rs`. Ampersands are special characters in XHTML that denote the start of a character entity, so have to be converted to entities themselves in order for your document to be valid. When passed to the synthetic speech engine, however, the entity will be converted back to the ampersand character. (In other words, the extra characters to encode the character will not affect the rendering.)

Single and double quote characters in X-SAMPA representations would similarly need to be escaped depending on the characters you use to enclose the attribute value.

It bears a quick note that the pronunciation in the `ssml:ph` attribute has to match the prose contained in the element it is attached to. By wrapping `span` elements around each individual word in this example, I've limited the translation of text to phonetic code to just the problematic words I want to fix. If I put the attribute on the parent `p` element, I'd have to transcode the entire sentence.

The upside of the granularity SSML markup provides should be clear now, though: you can overcome any problem no matter how small (or big) with greater precision than PLS files offer. The downside, of course, is having to work at the markup level to correct each instance that has to be overridden.

To hark back to the discussion of PLS files for a moment, though, we could further simplify the correction process by moving the more common pronunciation to our PLS lexicon and only fix the differing heteronym:

```
<lexicon
  version="1.0"
  alphabet="x-sampa"
  xml:lang="en"
  xmlns="http://www.w3.org/2005/01/pronunciation-lexicon">
  <lexeme>
    <grapheme>bass</grapheme>
    <phoneme>beIs</phoneme>
  </lexeme>
</lexicon>

<p>
  The guitarist was playing a bass that was shaped like a
  <span ssm1:alphabet="x-sampa" ssm1:ph="b&amp;s">bass</span>.
</p>
```

It's also not necessary to define the `ssml:alphabet` attribute every time. If we were only using a single alphabet throughout the document, which would be typical of most ebooks, we could instead define the alphabet once on the root `html` element:

```
<html ... ssm1:alphabet="x-sampa">
```

So long as the alphabet is defined on an ancestor of the element carrying the `ssml:ph` attribute, a rendering engine will interpret it correctly (and your document will be valid). (The root element is the ancestor of all the elements in the document, which is why these kinds of declarations are invariably found on it, in case you've ever wondered but were afraid to ask.)

Our markup can now be reduced to the much more legible and easily maintained:

```
<p>
  The guitarist was playing a bass that was shaped like a
  <span ssm1:ph="b&amp;s">bass</span>.
</p>
```



If you're planning to share content across ebooks or across content files within one, it's better to keep the attributes paired so that there is no confusion about which alphabet was used to define the pronunciation. It's not a common requirement, however.

But heteronyms are far from the only case for SSML. Any language construct that can be voiced differently depending on the context in which it is used is a candidate for SSML. Numbers are always problematic, as are weights and measures:

```
<p>
  There are <span
    ssm1:ph="w&quot;Vn T&quot;aUz@n t_hw&quot;En4i f&quot;0:r">1024</span> bits
    in a byte, not <span ssm1:ph="t_h&quot;En t_hw&quot;En4i f&quot;0:r">1024</span>,
    as the year is pronounced.
</p>
```

```
<p>
  It reached a high of <span
    ssm1:ph="'T3rti s&quot;Ev@n 'sEntI&quot;greId">37C</span> in the sun as I stood
    outside <span ssm1:ph="'T3rti s&quot;Ev@n si">37C</span> waiting for someone
    to answer my knocks and let me in.
</p>
```

```
<p>
  You'll be an <span ssm1:ph="Ekstr@ lArdZ">XL</span> by the end of Super Bowl
  <span ssm1:ph="'f0rti">XL</span> at the rate you're eating.
</p>
```

But there's unfortunately no simple guideline to give in terms of finding issues. It takes an eye for detail and an ear for possible different aural renderings. Editors and indexers are good starting resources for the process, as they should be able to quickly flag problem words during production so they don't have to be rooted out after the fact. Programs that can analyze books and report on potentially problematic words, although not generally available, are not just a fantasy. Their prevalence will hopefully grow now that EPUB 3 incorporates more facilities to enhance default renderings, as they can greatly reduce the human burden.

The only other requirement when using the SSML attributes that I haven't touched on is that you always have to declare the SSML namespace. I've omitted the declaration from the previous examples for clarity, and because the namespace is typically only specified once on the root `html` element as follows:

```
<html ... xmlns:ssm1="http://www.w3.org/2001/10/synthesis">
```

Similar to the `alphabet` attribute, we could have equally well attached the namespace declaration to each instance where we used the attributes:

```
<span
  xmlns:ssm1="http://www.w3.org/2001/10/synthesis"
  ssm1:ph="x-sampa"
  ...>
```

But that's a verbose approach to markup, and generally only makes sense when content is encapsulated and shared across documents, as I just noted, or expected to be extracted into foreign playback environments where the full document context is unavailable.

The question you may still be wondering at this point is what happens if a PLS file contains a pronunciation rule that matches a word that is also defined by an SSML pronunciation, how can you be sure which one wins? You don't have to worry, however, as the EPUB 3 specification defines a precedence rule that states that the SSML pronunciation must be honored. There'd be no way to override the global PLS definitions, otherwise, which would make SSML largely useless in resolving conflicts.

But to wrap up, a final note is that there is no reason why you couldn't make all your improvements in SSML. It's not the ideal way to tackle the problem, because of the text-level recognition and tagging it requires, at least in this author's opinion, but it may make more sense to internal production to only use a single technology and/or support for PLS may not prove universal (it's too early to know yet).

CSS3 Speech

You might be thinking the global definition power of PLS lexicons combined with the granular override abilities of SSML might be sufficient to cover all cases, so why a third technology? But you'd be only partly right.

The CSS3 Speech module is not about word pronunciation, however. It includes no phonetic capabilities, but defines how you can use CSS style sheet technology to control such aspects of synthetic speech rendering as the gender of voice to use, the amount of time to pause before and after elements, when to insert aural cues, etc.

The CSS3 Speech module also provides a simpler entry point for some basic voicing enhancements. The ability to write X-SAMPA or IPA pronunciations requires specialized knowledge, but the `speak-as` property masks the complexity for some common use cases.

You could use this property to mark all acronyms that are to be spelled out letter-by-letter, for example. If we added a class called 'spell' to the `abbr` elements we want spelled, as in the following example:

```
<abbr class="spell">IBM</abbr>
```

we could then define a CSS class to indicate that each letter should be voiced individually using the `spell-out` value:

```
.spell {  
  -epub-speak-as: spell-out  
}
```

It's no longer left to the rendering engine to determine whether the acronym is "wordy" enough to attempt to voice as a word now.



Note that the properties are all prefixed with “-epub-” because the Speech module was not a recommendation at the time that EPUB 3 was finalized. You must use this prefix until the Speech module is finalized and reading systems begin supporting the unprefixed versions.

The `speak-as` property provides the same functionality for numbers, ensuring they get spoken one digit at a time instead of as a single number, something engines will not typically do by default.

```
.digits {  
  -epub-speak-as: digits  
}
```

Adding this class to the following number would ensure that readers understand you're referring to the North American emergency line when listening to TTS playback:

```
<span class="digits">911</span>
```

The property also allows you to control whether or not to read out punctuation. Only some punctuation ever gets announced in normal playback, as it's generally used for pause effects, but you could require all punctuation to be voiced using the `literal-punctuation` value:

```
.punctuate {  
  -epub-speak-as: literal-punctuation  
}
```

This setting would be vital for grammar books, for example, where you would want the entire punctuation for each example to be read out to the student. Conversely, to turn punctuation off you'd use the `no-punctuation` value.

The `speak-as` property isn't a complex control mechanism, but definitely serves a useful role. Even if you are fluent with phonetic alphabets, there's a point where it doesn't make sense to have to define or write out every letter or number to ensure the engine doesn't do the wrong thing, and this is where the Speech module helps.

Where the module excels, however, is in providing playback control. But this is also an area where you may want to think twice before adding your own custom style sheet rules. Most reading systems typically have their own internal rules for playback so that the synthetic speech rendering doesn't come out as one long uninterrupted stream of monotone narration. When you add your own rules, you have the potential to interfere with the reader's default settings. But in the interests of thoroughness, we'll take a quick tour.

The first stop is the ability to insert pauses. Pauses are an integral part of the synthetic speech reading process, as they provide a non-verbal indication of change. Without them, it wouldn't always be clear if a new sentence were beginning or a new paragraph, or when one section ends and another begins.

The CSS3 Speech module includes a `pause` property that allows you to control the duration to pause before and after any element. For example, we could define a half-second pause before headings followed by a quarter-second pause after by including the following rule:

```
h1 {  
  -epub-pause: 50ms 25ms  
}
```

Aural cues are equally helpful when it comes to identifying new headings, as the pause alone may not be interpreted by the listener as you expect. The Speech module includes a `cue` property for exactly this purpose:

```
h1 {
  -epub-pause: 50ms 25ms;
  -epub-cue: url('audio/ping.mp3') none
}
```

(Note that the addition of the `none` value after the audio file location. If omitted, the cue would also sound after the heading was finished.)

And finally, the `rest` property provides fine-grained control when using cues. Cues occur after any initial pause before the element (as defined by the `pause` property), and before any pause after. But you may still want to control the pause that occurs between the cue sounding and the text being read and between the end of the text being read and the trailing cue sounding (i.e., so that the sound and the text aren't run together). The `rest` property is how you control the duration of these pauses.

We could update our previous example to add a 10 millisecond rest after the cue is sounded to prevent run-ins as follows:

```
h1 {
  -epub-pause: 50ms 25ms;
  -epub-cue: url('audio/ping.mp3') none;
  -epub-rest: 10ms 0ms
}
```

But again, if I didn't say it forcefully enough earlier, it's best not to tweak these properties unless you're targeting a specific user group, know their needs, and know that their players will not provide sufficient quality "out of the box." Tread lightly, in other words.

A final property, that is slightly more of an aside, is `voice-family`. Although not specifically accessibility related, it can provide a more flavorful synthesis experience for your readers.

If your ebook contains dialogue, or the gender of the narrator is important, you can use this property to specify the appropriate gender voice. We could set a female narrator as follows:

```
body {
  -epub-voice-family: female
}
```

and a male class to use as needed for male characters:

```
.male {
  -epub-voice-family: male
}
```

If we added these rules to a copy of *Alice's Adventures in Wonderland*, we could now differentiate the Cheshire Cat using the male voice as follows:

```

<p>
  Alice: But I don't want to go among mad people.
</p>

<p class="male">
  The Cat: Oh, you can't help that.
  We're all mad here. I'm mad. You're mad.
</p>

```

You can also specify different voices within the specified gender. For example, if a reading system had two male voices available, you could add some variety to the characters as follows by indicating the number of the voice to use:

```

.first-male {
  -epub-voice-family: male 1
}

.second-male {
  -epub-voice-family: male 2
}

```

At worst, the reading system will ignore your instruction and only present whatever voice it has available, but this property gives you the ability to be more creative with your text-to-speech renderings for those systems that do provide better support.



The CSS3 Speech module contains more properties than I've covered here, but reading systems are only required to implement the limited set of features described in this section. You may use the additional properties the module makes available (e.g., pitch and loudness control), but if you do your content may not render uniformly across platforms. Carefully consider using innovative or disruptive features in your content, as this may hinder interoperability across reading systems.

Whatever properties you decide to add, it is always good practice to separate them into their own style sheet. You should also define them as applicable only for synthetic speech playback using a media at-rule as follows:

```

@media speech {

  .spell {
    -epub-speak-as: spell-out
  }

}

```

As I noted earlier, reading systems will typically have their own defaults, and separating your aural instructions will allow them to be ignored and/or turned off on systems where they're unwanted.

For completeness, you should also indicate the correct media type for the style sheet when linking from your content document:

```
<link rel="stylesheet" media="speech" href="synth.css" />
```

And that covers the full range of synthetic speech enhancements. You now have a whole arsenal at your disposal to create high-quality synthetic speech.

The Coded Word: Scripted Interactivity

Whether you're a fan of scripted ebooks or not, EPUB 3 has opened the door to their creation, so we'll now take a look at some of the potential accessibility pitfalls and how they can be avoided.

One of the key new terms you'll hear in relation to the use of scripting in EPUB 3 is *progressive enhancement*. The concept of progressive enhancement is not original to EPUB, however, nor is it limited to scripting. I've actually been making a case for many of its other core tenets throughout this guide, such as separation of content and style, content conveying meaning, etc. Applied in this context, however, it means that scripting must only enhance your core content.

We've already covered why structure and semantics should carry all the information necessary to understand your content, but that presupposes that it is all available. The ability for scripts to remove access to content from anyone without a JavaScript-enabled reading system is a major concern not just for persons using accessible devices, but for all readers.

And that's why scripting access to content [is forbidden in EPUB 3](#). If you try to circumvent the specification requirement and treat progressive enhancement as just an "accessibility thing," you're underestimating the readership that are going to rely on your content rendering properly without scripting. Picture buying a book that has pages glued together and you'll get an idea of how excited your readers will be that you thought no one would notice.



Note that it's not a truism that you can expect JavaScript support in EPUB 3 reading systems. There will undoubtedly be widespread support for scripting in time, but support is an optional feature that vendors and developers can choose to ignore.

Meeting the general requirement to keep your text accessible is really not asking a lot, though. As soon as you turn to JavaScript to alter (or enable) access to prose, you should realize you're on the wrong path. To this end:

- Don't include content that can only be accessed (made visible) through scripted interaction.
- Don't script-enable content based on a reader's preferences, location, language, or any other setting.

- Don't require scripting in order for the reader to continue moving through the content (e.g., choose your own adventure books).

Whether or not your prose can be accessed is not hard to test, even if it can't be done reliably by validators like [epubcheck](#). Turn off JavaScript and see if you can navigate your document from beginning to end. You may not get all the bells and whistles when scripting is turned off, but you should be able to get through with no loss of information. If you can't, you need to review why prose is not available or has disappeared, why navigation gets blocked, etc., and find another way to do what you were attempting.

Don't worry that this requirement means all the potential scripting fun is being taken out of ebooks, though. Games and puzzles and animations and quizzes and any other secondary content you can think of that requires scripting are all fair game for inclusion. But when it comes to including these there are two considerations to make, very similar to choosing when to describe images:

- Does the scripted content you're embedding include information that will be useful to the reader's comprehension (demos, etc.), or is it included purely for pleasure (games)?
- Can the content be made accessible in a usable way and can you provide a fallback alternative that provides the same or similar experience?

The answer to the first question will have some influence how you tackle the second. If the scripted content provides information that the reader would otherwise not be able to obtain from the prose, you should consider other alternative forms for making that information available, for example:

- If you script an interactive demo using the `canvas` element, consider also providing a transcript of the information for readers who may not be able to interact with it.
- If you're including an interactive form that automatically evaluates the reader's responses, also include access to an answer key.
- If you're adding a problem or puzzle to solve, also provide the solution so the reader can still learn the steps to its completion.

None of the above suggestions are intended to remove the responsibility to try and make the content accessible in the first place, though. Scripting of accessible forms, for example, should be a trivial task for developers familiar with WAI-ARIA (we'll look at some practices in the coming section). But trivial or not, because scripting will not necessarily be available, it's imperative that you provide other means for readers to obtain the full experience.

If the scripted content is purely for entertainment purposes, however, create a fallback commensurate with the value of that content to the overall ebook (if it absolutely cannot be made accessible natively). Like decorative images, a reader unable to interact with non-essential content is not going to be hugely interested in reading a five-page dissertation on each level of your game. A simple idea of what it does will usually suffice.

A Little Help: WAI-ARIA

Although fallbacks are useful when scripting is not available, you should still aim to make your scripted content accessible to all readers. Enter the W3C Web Accessibility Initiative's (WAI) [Accessible Rich Internet Application \(ARIA\)](#) specification.

The technology defined in this specification can be used in many situations to improve content accessibility. We've already encountered the `aria-describedby` attribute in looking at how to add descriptions and summaries, for example.

I'm now going to pick out three common cases for scripting to further explore how ARIA can enhance the accessibility of EPUBs: custom controls, forms, and live regions.

Custom Controls

Custom controls are not standard form elements that you stylize to suit your needs, just to be clear. Those are the good kinds of custom controls—if you want to call them custom—as they retain their inherent accessibility traits whatever you style them to look like. Readers will not have problems interacting with these controls as they natively map to the underlying accessibility APIs, and so will work regardless of the scripting capabilities any reading system has built in.

A custom control is the product of taking an HTML element and enhancing it with script to emulate a standard control, or building up a number of elements for the same purpose. Using images to simulate buttons is one of the more common examples, as custom toolbars are often created in this way. There is typically no native way for a reader using an accessible device to interact with these kinds of custom controls, however, as they are presented to them as whatever HTML element was used in their creation (e.g., just another `img` element in the case of image buttons).

It would be ideal if no one used custom controls, and you should try to avoid them unless you have no other choice, but the existence of ARIA reflects the reality that these controls are also ubiquitous. The increase in native control types in HTML5 holds out hope for a reduction in their use, but it would be neglectful not to cover some of the basics of their accessible creation. Before launching out on your own, it's good to know what you're getting into.



There are widely available toolkits, like [jQuery](#), that bake ARIA accessibility into many of the custom widgets they allow you to create. You should consider using these if you don't have a background in creating accessible controls.

If you aren't familiar with ARIA, a very quick, high-level introduction for custom controls is that it provides a map between the new control and the standard behaviors of the one being emulated (e.g., allowing your otherwise-inaccessible image to function

identically to the `button` element as far as the reader is concerned). This mapping is critical, as it's what allows the reader to interact with your controls through the underlying accessibility API. (The ARIA specification includes a [graphical depiction](#) that can help visualize this process.)

Or, put differently, ARIA is what allows the HTML element you use as a control to be identified as what it represents (`button`) instead of what it is (`image`). It also provides a set of attributes that can be set and controlled by script to make interaction with the control accessible to all. As the reader manipulates your now-identifiable control, the changes you make to these attributes in response get passed back to the underlying accessibility API. That in turn allows the reading system or assistive technology to relay the new state on to the reader, completing the cycle of interaction.

But to get more specific, the role an element plays is defined by attaching the ARIA `role` element to it. The following is a small selection of the [available role values](#) you can use in this attribute:

- `alert`
- `button`
- `checkbox`
- `dialog`
- `menuitem`
- `progressbar`
- `radio`
- `scrollbar`
- `tab`
- `tree`

Here's how we could now use this attribute to define an image as a audio clip start button:

```

```

Identifying the role is the easy part, though. Just as standard form controls have states and properties that are controlled by the reading system, so too must you add and maintain these on any custom controls you create.

A `state, to clarify,` tells you something about the nature of the control at a given point in time: if an element represents a checkable item, for example, its current state will either be checked or unchecked; if it can be hidden, its state may be either hidden or visible; if it's collapsible, it could be expanded or collapsed; and so on.

Properties, on the other hand, typically provide meta information about the control: how to find its label, how to find a description for it, its position in a set, etc.

States and properties are both expressed in ARIA using attributes. For example, the list of available states currently includes all of the following:

- aria-busy
- aria-checked
- aria-disabled
- aria-expanded
- aria-grabbed
- aria-hidden
- aria-invalid
- aria-pressed
- aria-selected

The list of available properties is much larger, but a small sampling includes:

- aria-activedescendant
- aria-controls
- aria-describedby
- aria-dropeffect
- aria-flowto
- aria-labelledby
- aria-live
- aria-posinset
- aria-required



See [section 6.6](#) of the ARIA specification for a complete list of all states and properties, including definitions.

All of these state and property attributes are supported in EPUB 3 content documents, and their proper application and updating as your controls are interacted with is how the needed information gets relayed back to the reader. (Note: you only have to maintain their values; you don't have to worry about the underlying process that identifies the change and passes it on.)

The natural question at this point is which states and properties do you have to set when creating a custom control. It would be great if there were a simple chart that could be followed, but unfortunately the ones that you apply is very much dependent on the

type of control you're creating, and what you're using it to do. To be fully accessible, you need to consider all the ways in which a reader will be interacting with your control, and how the states and properties need to be modified to reflect the reality of the control as each action is performed. There is no one-size-fits-all solution, in other words.



To see which properties and states are supported by the type of control you're creating, refer to the [role definitions](#) in the specification. Knowing what you can apply is helpful in narrowing down what you need to apply.

If you don't set the states and properties, or set them incorrectly, it follows that you'll impair the ability of the reader to access your content. Implementing them badly can be just as frustrating for a reader as not implementing them at all, too. You could, for example, leave the reader unable to start your audio clip, unable to stop it, stuck with volume controls that only go louder or softer, etc. Their only recourse will be shutting down their ebook and starting over.

These are the accessibility pitfalls you have to be aware of when you roll your own solutions. Some will be obvious, like a button failing to initiate playback, but others will be more subtle and not caught without extensive testing, which is also why you should engage the accessibility community in checking your content.

But let's take a look at some of the possible issues involved in maintaining states. Have a look at the following much-reduced example of list items used to control volume:

```
<ul>
  <li role="button"
      tabindex="0"
      onclick="increaseVolume('audio01')>Louder</li>

  <li role="button"
      tabindex="0"
      onclick="decreaseVolume('audio01')>Softer</li>
</ul>
```

This setup looks simple, as it omits any states or properties at the outset, but now let's consider it in the context of a real-world usage scenario. As the reader increases the volume, you'll naturally be checking whether the peak has been reached in order to disable the control. With a standard button, when the reader reached the maximum volume you'd just set the button to be disabled with a line of JavaScript; the button gets grayed out for readers and is marked as disabled for the accessibility API. Nice and simple.

List items can't be natively disabled, however (it just doesn't make any sense, since they aren't expected to be active in the first place). You instead have to set the `aria-disabled` attribute on the list item to identify the change to the accessibility API, remove the event that calls the JavaScript (as anyone could still activate and fire the referenced

code if you don't), and give sighted readers a visual effect to indicate that the button is no longer active.

Likewise, when the reader decreases the volume from the max setting, you need to re-enable the control, re-add the `onClick` event, and re-style the option as active. The same scenario plays out when the reader hits the bottom of the range for the volume decrease button.

In other words, instead of having to focus only on the logic of your application, you now also have to focus on all the interactions with your custom controls. This extra programming burden is why rolling your own was not recommended at the outset. This is a simple example, too. The more controls you add, the more complex the process becomes and the more potential side-effects you have to consider and account for.

If you still want to pursue your own controls, though, or just want to learn more, the [Illinois Center for Information Technology and Web Accessibility](#) maintains a comprehensive set of examples, with working code, that are worth the time to review. You'll discover much more from their many examples than I could reproduce here. The ARIA authoring practices guide also walks through the process of [creating an accessible control](#).

A quick note on `tabindex` is also in order, as you no doubt noticed it on the preceding examples. Although this is actually an HTML attribute, it goes hand-in-hand with ARIA and custom controls because it allows you to specify additional elements that can receive keyboard focus, as well the order in which all elements are traversed (i.e., it improves keyboard accessibility). It is critical that you add the attribute to your custom controls, otherwise readers won't be able to navigate to them.



What elements a reader can access by the keyboard by default is reading system-dependent, but typically only links, form elements, and multimedia and other interactive elements receive focus by default. Keep this in mind when you roll your own controls, otherwise readers may not have access to them.

Here's another look at our earlier image button again:

```

```

By adding the attribute with the value `0`, we've enabled direct keyboard access to this `img` element. The `0` value indicates that we aren't giving this control any special significance within the document, which is the default for all elements that can be natively tabbed to. To create a tab order, we could assign incrementing positive integers to the controls, but be aware that this can affect the navigation of your document, as all elements with a positive `tabindex` value are traversed before those set to `0` or not specified

at all (in other words, don't add the value 1 because to you it's the first element in your control set).

In many situations, too, a single control would not be made directly accessible. The element that contains all the controls would be the accessible element, as in the following example:

```
<div role="group" tabindex="0">
  <img role="button" ... />
  <img role="button" ... />
</div>
```

Access to the individual controls inside the grouping `div` would be script-enabled. This would allow the reader to quickly skip past the control set if they aren't interested in what it does (otherwise they would have to tab through every control inside it).



See the HTML5 specification for more information on [how this attribute works](#).

A last note for this section concerns event handlers. Events are what are used to trigger script actions (`onclick`, `onblur`, etc.). How you wire up your events can impact on the ability of the reader to access your controls, and can lead to keyboard traps (i.e., the inability to leave the control), so you need to pay attention to how you add them.

We could add an `onclick` event to our image button to start playback as follows:

```

```

But, if we'd accidentally forgotten the `tabindex` attribute, a reader navigating by keyboard would not have been able to find or access this control. Even though `onclick` is considered a device-independent event, if the reader cannot reach the element they cannot use the Enter key to activate it, effectively hiding the functionality from them.

You should always ensure that actions can be triggered in a device-independent manner, even if that means repeating your script call in more than one event type. Don't rely on any of your readers using a mouse, for example.

But again, it pays to engage people who can test your content in real-world scenarios to help discover these issues than to hope you've thought of everything.

Forms

Having covered how to create custom controls, we'll now turn to forms, which are another common problem area ARIA helps address. To repeat myself for a moment, though, the first best practice when creating forms is to always use the native form elements that HTML5 provides. See the last section again for why rolling your own is not a good idea.

When it comes to implementing forms, the logical ordering of elements is one key to simplifying access and comprehension. The use of `tabindex` can help to correct navigation, as we just covered, but it's better to ensure your form is logically navigable in the first place. Group form fields and their labels together when you can, or place them immediately next to each other so that one always follows the other in the reading order.

And always clearly identify the purpose of form fields using the `label` element. You should also always add the new HTML5 `for` attribute so that the labels can be located regardless of how the reader enters the field or where they are located in the document markup. This attribute identifies the `id` of the form element the `label` element labels:

```
<label id="fname-label" for="fname">First name:</label>

<input type="text"
      id="fname"
      name="first-name"
      aria-labelledby="fname-label" />
```

I've also added the `aria-labelledby` attribute to the `input` element in this example to ensure maximum compatibility across systems, but its use is critical if your form field is not identified by a `label` element (only `label` takes the `for` attribute). As the `label` element can be used in just about every element that can carry a label, there's little good reason to omit using it.

For example, if you have to use a table to lay out your form, don't be lazy and use table cells alone to convey meaning:

```
<table>
  <tr>
    <td>
      <label id="fname-label" for="fname">First name:</label>
    </td>
    <td>
      <input type="text"
            id="fname"
            name="first-name"
            aria-labelledby="fname-label" />
    </td>
  </tr>
  ...
</table>
```

Note that you also should include the `for` attribute regardless of whether the `label` precedes, follows or includes the form field.

Another pain point comes when a reader fills in a form only to discover after the fact that you had special instructions they were supposed to follow. When specifying entry requirements for completing the field, include them within the `label` or attach an `aria-describedby` attribute so that the reader can be informed right away:

```
<label for="username-label">User name:</label>

<input type="text"
      id="uname"
      name="username"
      aria-labelledby="username-label"
      aria-describedby="username-req" />

<span id="username-req">User names must be between 8 and 16 characters in length and
contain only alphanumeric characters.</span>
```

The new HTML5 `pattern` attribute can also be used to improve field completion. If your field accepts regular expression-validatable data, you can add this attribute to automatically check the input. When using this attribute, the HTML5 specification recommends the restriction be added to the `title` attribute.

We could reformulate our previous example now as follows:

```
<input type="text"
      id="uname"
      name="username"
      aria-labelledby="username-label"
      pattern="[A-Za-z0-9]{8,16}"
      title="Enter a user name between 8 and 16 characters in length
and containing only alphanumeric characters" />
```

Another common nuisance in web forms of old has been the use of asterisks and similar text markers and visual cues to indicate when a field was required, as there was no native way to indicate the requirement to complete. These markers were not always identifiable by persons using assistive technologies. HTML5 now includes the `required` attribute to cover this need, however. ARIA also includes a `required` attribute of its own. Similar to labeling, it's a good practice at this time to add both to ensure maximum compatibility:

```
<input type="text"
      id="uname"
      name="username"
      aria-labelledby="username-label"
      pattern="[A-Za-z0-9]{8,16}"
      title="Enter a user name between 8 and 16 characters in length
and containing only alphanumeric characters"
      required="required"
      aria-required="true" />
```

An accessible reading system could now announce to the reader that the field is required when the reader enters it. Adding a clear prose indication that the field is required to the `label` is still good practice, too, as colors and symbols are never a reliable or accessible means of conveying information:

```
<label for="uname">User name: (required)</label>
```

ARIA also includes a property for setting the validity of an entry field. If the reader enters invalid data, you can set the `aria-invalid` property in your code so that the reading system can easily identify and move the reader to the incorrect field. For example, your scripted validation might include the following line to set this state when the input doesn't pass your tests:

```
document.getElementById('address').setAttribute('aria-invalid', true);
```

Note, however, that you must not set this state by default; no data entered does not indicate either validity or invalidity.

In addition to labeling individual form fields, you should also group and identify any regions within your form (a common example on the web is forms with separate fields for billing and shipping information). The traditional HTML `fieldset` element and its child `legend` element cover this need without special attributes.

So, to try and sum up, the best advice with forms is to strive to make them as accessible as you can natively (good markup and logical order), but not to forget that WAI-ARIA exists and has a number of useful properties and states that can enhance your forms to make them more accessible.

Live Regions

Although manipulating the prose in your ebook by script is forbidden, it doesn't mean you can't dynamically insert or modify any text. Automatically displaying the result of a quiz or displaying the result of a calculation are a just a couple of examples of cases where dynamic prose updates would legitimately be useful for readers. You may also want to provide informative updates, such as the number of characters remaining in an entry field.

The problem with these kinds of dynamic updates is how they're made available to readers using accessible technologies. When you update the main document by rewriting the inner text or html of an element, how that change gets reported to the accessible technology, if at all, is out of your control in plain old HTML.

The update could force the reader to lose their place and listen to the changed region every time, or it could be ignored entirely. ARIA has solved this problem with the introduction of live regions, however.

If you're going to use an element to insert dynamic text, you mark this purpose by attaching an `aria-live` attribute to it. The value of this attribute also tells an assistive technology how to present the update to the reader. If you set the value `polite`, for example, the assistive technology will wait until an idle period before announcing the change (e.g., after the user is done typing for character counts). If you set it to `assertive`, the reading system will announce the change immediately (e.g., for results that the reader is waiting on).

You could set up a simple element to write results to with no more code than follows:

```
<div id="result" aria-live="assertive"/>
```

Now when you write using the `innerHTML` property, the new text will be read out immediately. Be careful when using the `assertive` setting, however. You can annoy your readers if their system blurts out every inconsequential change you might happen to write as it happens.

If you write out results a bit at a time, or need to update different elements within the region, the `aria-busy` attribute should be set to `true` before the first write to indicate to the reading system that the update is in progress. If you don't, the reading system will announce the changes as you write them. So long as the state is marked as busy (`true`), however, the reading system will wait for the state to be changed backed to `false` before making any announcement.

You should also take care about how much information you inform the reader of. If you're updating only small bits of text, the reading system might only announce the new text, leaving the reader confused about what is going on. Conversely, you might add a new node to a long list, but the reader might be forced to listen to all the entries that came before it again, depending on how you have coded your application.

The `aria-atomic` attribute gives you control over the amount of text that gets announced. If you set it to `true`, for a region, all the content will be read whenever you make a change inside it. For example, if you set a paragraph as live and add this attribute, then change the text in a `span` inside it, the entire paragraph will be read. In this example:

```
<p aria-live="true" aria-atomic="true">  
  Your current BMI is: <span id="result"/>  
</p>
```

Writing the reader's body mass index value to the embedded `span` will cause the whole text to be read. If you set the attribute to `false` (or omit it), only the prose in the element containing the text change gets announced. Using our last example, only the body mass index value in isolation would be announced.

You can further control this behavior by also attaching the `aria-relevant` attribute. This attribute allows you to specify, for example, that all node changes in the region should be announced, only new node additions, or only deletions (e.g., for including data feeds). It can also be set to only identify text changes. You can even combine values (the default is `additions text`).

We could use these attributes to set up a fictional author update box using an ordered list as follows:

```
<p id="feed-label">What's the Author Saying...</p>  
<ol id="feed"  
  aria-live="polite"  
  aria-atomic="true"  
  aria-relevant="additions"
```

```
aria-labelledby="feed-label">
...
</ol>
<a href="http://www.example.com/authorsonline">Go online to view</a>
```

Only the new list items added for each incoming message will be read now. The old messages we pull out will disappear silently. (And I've also added a traditional link out for anyone who doesn't have scripting enabled!)

There are also special roles that automatically identify a region as live. Instead of using the `aria-live` attribute to indicate our results field, we could have instead set up an alert region as follows:

```
<div role="alert" id="results"/>
```

The following roles are also treated as indicating live regions: `marquee`, `log`, `status`, and `timer`.

And that's a quick run-through of how to ensure that all readers get alerted of changes you make to the content. It's not a complicated process, but you need to remember to ensure that you set these regions otherwise a segment of your readers will not get your updates.



My hope is these sections have given you an easy introduction to ARIA and the features it provides to make EPUB content accessible

For additional information, some good starting points include: the coverage given in [Universal Design for Web Applications](#) by Wendy Chisholm and Matt May (also an excellent guide to accessible Web content development); Gez Lemon's [introduction to creating rich applications](#); and, of course, the [authoring practices guide](#) that accompanies the ARIA specification.

A Blank Slate: Canvas

Another anticipated use for scripting is to automate the new HTML5 `canvas` element. This element provides an automatable surface for drawing on, whether it's done by the content creator (games, animations, etc.) or the reader (drawing or writing surface), which is why I omitted tackling it with the rest of the semantics and structure elements.

Although a potentially interesting element to use in ebooks, at this time the `canvas` element remains largely a black hole to assistive technologies. A [summary of the discussions](#) that have been taking place to fix the accessibility problems as of writing is available on the Paciello Group website. Fixes for these accessibility issues will undoubtedly come in time, perhaps directly for the element or perhaps through WAI-ARIA, but it's too soon to say.

So is the answer to avoid the element completely until the problems are solved? It would be nice if you could, but wouldn't be realistic to expect of everyone. Using it judiciously would be a better course to steer.

For now including accessible alternatives is about all you can do. If you're using the element to draw graphs and charts, you could add a description with the data using the `aria-describedby` attribute and the techniques we outlined while dealing with images. If you're using the element for games and the like, consider the issues we detailed at the outset of the section in determining how much information to give.

With `canvas`, we really have to wait and see, unfortunately.

Conclusion

EPUB 3 holds out much promise, but only if you care about the quality of your content and actively work to make it better. If I've done my job, though, accessibility is hopefully no longer a foreign concept or impossible-sounding ideal anymore. It's fundamentally about high-quality data, with hooks in for people who can't consume the content in its native format, whether auditory or visual.

The people you need to produce accessible EPUBs are not hard to find, either. Web content developers abound as the internet generation comes of age. And unlike in the early dark days of web accessibility, more and more people are learning WCAG and WAI-ARIA guidelines for accessible production. They're not skills you can ignore as a developer, as so much basic accessibility legislation is premised on them now.

My point, however, is only that creating accessible EPUB 3 publications is not a costly proposition. It doesn't require seeking out highly-specialized skills that won't provide you a return on your investment. Reflowable web content is the direction publishing is heading in, and well down the road to.

But to wrap up, no guide can ever make you take action, only impart some measure of knowledge. Assuming I've met that threshold, the onus is now on you to take what you've learned and put it to good use.

EPUB 3 Best Practices Teaser

Accessible EPUB 3 is an excerpt of the book *EPUB 3 Best Practices*, currently in development for publication in 2012.

For more details and updates on its anticipated release date, keep an eye on the web page for the book:

<http://shop.oreilly.com/product/0636920024897.do>

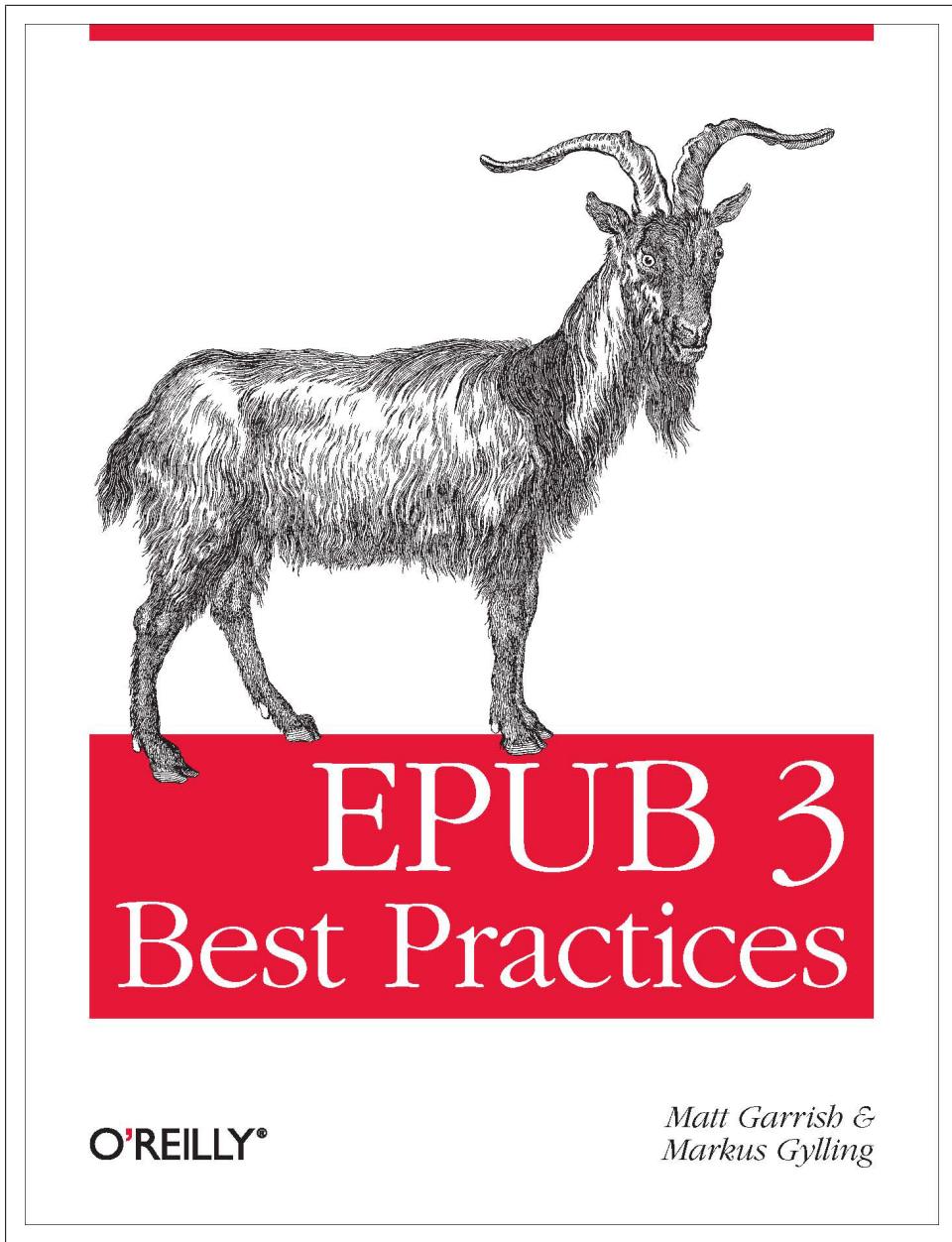


Figure 4-1. EPUB 3 Best Practices, coming in 2012

About the Book

The new EPUB 3 specification from the IDPF incorporates a wide range of technologies and functionality that are set to revolutionize electronic publishing. The format is poised to make the static two-dimensional page a thing of the past, introducing the world to new rich multimedia reading experiences and scripted interactivity. But a specification that offers so much can be a daunting thing to learn.

EPUB 3 Best Practices steps in to help fill the knowledge void. Authored by people involved in the development of the specification, and with extensive experience in electronic publishing, this guide will provide you with a solid foundation on which to begin developing your own EPUBs. Topics covered include:

- A comprehensive survey of accessible production features and best practices
- A walkthrough of the new global language support features
- An introduction to the new multimedia elements and how to use them to embed content
- A guide to best practices for authoring of interactive elements and scripting
- A review of publication and distribution metadata
- Techniques for fixed and adaptive layouts

EPUB 3 Best Practices is a must-read for anyone looking to unleash the potential of the new format.

About the Author

Matt Garrish lives and works in Toronto where he does what he can to help bridge the print divide that sadly still keeps much of the world's literature and information from being available to everyone. He's worked closely with CNIB and the DAISY Consortium in their efforts to make the world a more accessible place—including editing the Z39.98 Authoring and Interchange specification—and drew on his years of experience ripping the guts out of EPUBs to make braille when invited to work as the editor of the EPUB3 revision. He is the author of *What is EPUB3?*.

