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">dɪ'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&s. 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 ssml:alphabet="x-sampa" ssml: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 … ssml: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 ssml: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
    ssml:ph="w&amp;quot;Vn T&amp;quot;aUz@n t_hw&amp;quot;En4i f&amp;quot;O:r">1024</span> bits
    in a byte, not <span ssml:ph="t_h&amp;quot;En t_hw&amp;quot;En4i f&amp;quot;O:r">1024</span>,
    as the year is pronounced.
</p>

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

<p>
    You'll be an <span ssml:ph="Ekstr@ lArdZ">XL</span> by the end of Super Bowl
    <span ssml:ph="'fOrti">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:ssml="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:ssml="http://www.w3.org/2001/10/synthesis"
    ssml: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:

```
<img src="controls/start.png"
        id="audio-start"
        role="button"
        tabindex="0"
        alt="Start"/>
```

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:

```
<img src="controls/start.png"
        id="audio-start"
        alt="Start"
        role="button"
        tabindex="0"/>
```

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:

```
<img src="controls/start.png"
        id="audio-start"
        alt="Start"
        role="button"
        tabindex="0"
        onclick="startPlayback('audio01')"/>
```

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:

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.

CHAPTER 4
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

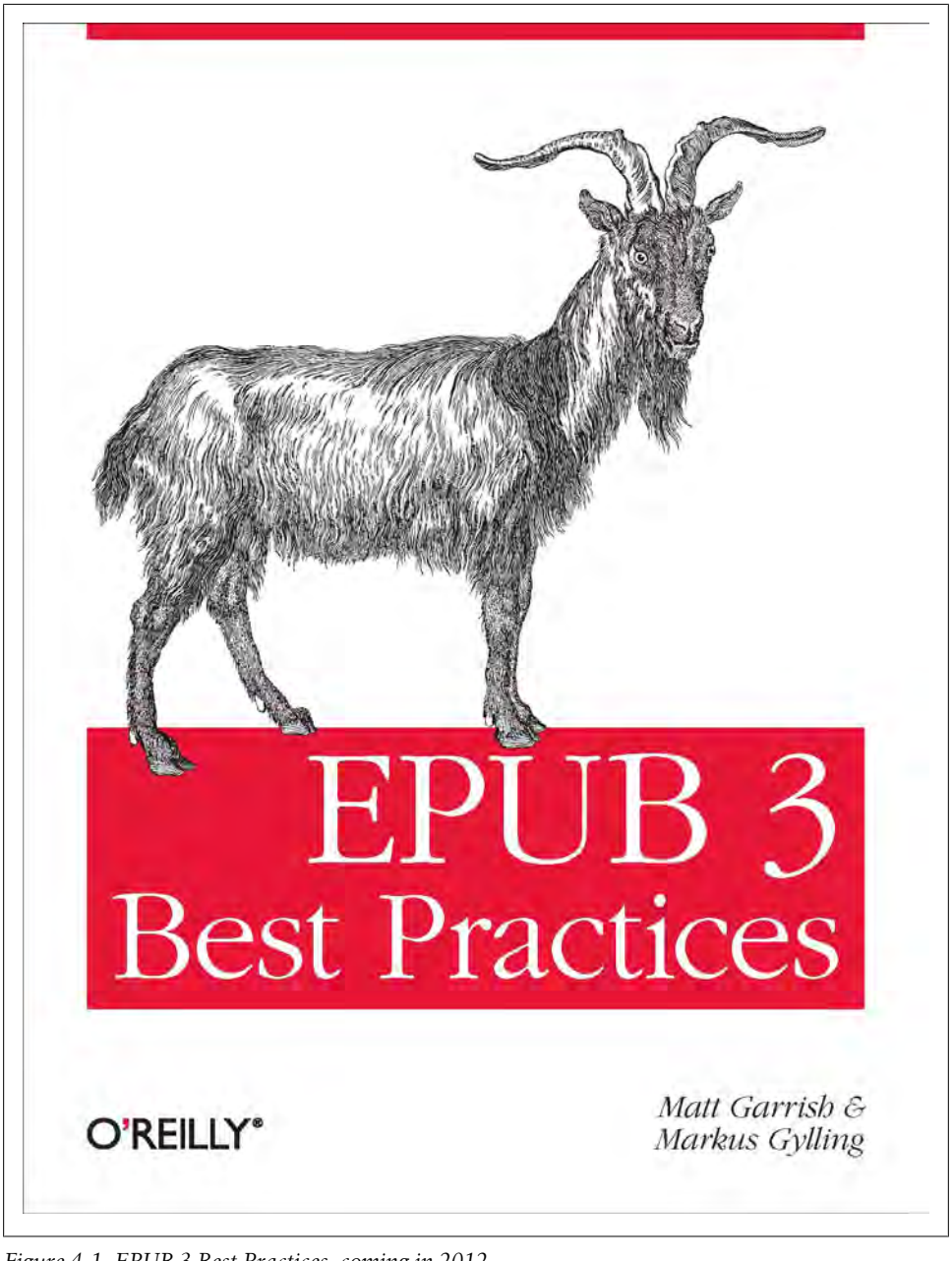

Figure 4-1. EPUB 3 Best Practices, coming in 2012

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?*.