



HTML 5.2

W3C Recommendation, 14
December 2017

§ 1. Introduction

§ 1.1. Background

This section is non-normative.

HTML is the World Wide Web's core markup language. Originally, HTML was primarily designed as a language for semantically describing scientific documents. Its general design, however, has enabled it to be adapted, over the subsequent years, to describe a number of other types of documents and even applications.

§ 1.2. Audience

This section is non-normative.

This specification is intended for authors of documents and scripts that use the features defined in this specification, implementors of tools that operate on pages that use the features defined in this specification, and individuals wishing to establish the correctness of documents or implementations with respect to the requirements of this specification.

This document is probably not suited to readers who do not already have at least a passing familiarity with Web technologies, as in places it sacrifices clarity for precision, and brevity for completeness. More approachable tutorials and authoring guides can provide a gentler introduction to the topic.

In particular, familiarity with the basics

§ 1.3. Scope

The scope of this specification is not to describe an entire operating system. In particular, hardware configuration software, image manipulation tools, and applications that users would be expected to use with high-end workstations on a daily basis are out of scope. In terms of applications, this specification is targeted specifically at applications that would be expected to be used by users on an occasional basis, or regularly but from disparate locations, with low CPU requirements. Examples of such applications include online purchasing systems, searching systems, games (especially multiplayer online games), public telephone books or address books, communications software (e-mail clients, instant messaging clients, discussion software), document editing software, etc.

§ 1.4. History

This section is non-normative.

For its first five years (1990-1995), HTML went through a number of revisions and experienced a number of extensions, primarily hosted first at CERN, and then at the IETF.

With the creation of the W3C, HTML's development changed venue again. A first abortive attempt at extending HTML in 1995 known as HTML 3.0 then made way to a more pragmatic approach known as HTML 3.2, which was completed in 1997. HTML 4.01 quickly followed later that same year.

The following year, the W3C membership decided to stop evolving HTML and instead begin work on an XML-based equivalent, called XHTML. This effort started with a reformulation of HTML 4.01 in XML, known as XHTML 1.0, which added no new features except the new serialization, and which was completed in 2000. After XHTML 1.0, the W3C's focus turned to making it easier for other working groups to extend XHTML, under the banner of XHTML Modularization. In parallel with this, the W3C also worked on a new language that was not compatible with the earlier HTML and XHTML languages, calling it XHTML 2.0.

Around the time that HTML's evolution was stopped in 1998, parts of the API for HTML developed by browser vendors were specified and published under the name DOM Level 1 (in 1998) and DOM Level 2 Core and DOM Level 2 HTML (starting in 2000 and culminating in 2003). These efforts then petered out, with some DOM Level 3 specifications published in 2004 but the working group being closed before all the Level 3 drafts were completed.

In 2003, the publication of XForms, a technology which was positioned as the

Shortly thereafter, Apple, Mozilla, and Opera jointly announced their intent to continue working on the effort under the umbrella of a new venue called the WHATWG. A public mailing list was created, and the draft was moved to the WHATWG site. The copyright was subsequently amended to be jointly owned by all three vendors, and to allow reuse of the specification.

The latter requirement in particular required that the scope of the HTML specification include what had previously been specified in three separate documents: HTML 4.01, XHTML 1.1, and DOM Level 2 HTML. It also meant including significantly more detail than had previously been considered the norm.

In 2006, the W3C indicated an interest to participate in the development of HTML 5.0 after all, and in 2007 formed a working group chartered to work with the WHATWG on the development of the HTML specification. Apple, Mozilla, and Opera allowed the W3C to publish the specification under the W3C copyright, while keeping a version with the less restrictive license on the WHATWG site.

For a number of years, both groups then worked together under the same editor: Ian Hickson. In 2011, the groups came to the conclusion that they had different goals: the W3C wanted to draw a line in the sand for features for a HTML 5.0 Recommendation, while the WHATWG wanted to continue working on a Living Standard for HTML, continuously maintaining the specification and adding new features. In mid 2012, a new editing team was introduced at the W3C to take care of creating a HTML 5.0 Recommendation and prepare a Working Draft for the next HTML version.

Since then, the W3C Web Platform WG has been cherry picking patches from the WHATWG that resolved bugs

A separate document is published to document the differences between the HTML specified in this document and the language described in the HTML 4.01 specification. [\[HTML5-DIFF\]](#)

This section is non-normative.

HTML, its supporting DOM APIs, as well as many of its supporting technologies, have been developed over a period of several decades by a wide array of people with different priorities who, in many cases, did not know of each other's existence.

Despite all this, efforts have been made to adhere to certain design goals. These are described in the next few subsections.

This section is non-normative.

1.5.2. Compliance with other specifications

This specification interacts with and relies on a wide variety of other specifications. In certain circumstances, unfortunately, conflicting needs have led to this specification violating the requirements of these other specifications. Whenever this has occurred, the transgressions have each been noted as a "**willful violation**", and the reason for the violation has been noted.

This section is non-normative.

- Authors can use the `class` attribute to extend elements, effectively creating their own elements, while using the most applicable existing "real" HTML element, so that browsers and other tools that don't know of the extension can still support it somewhat well. This is the tack used by microformats, for example.

- ## § 1.6. HTML vs XML Syntax

This specification defines an abstract language for describing documents and applications, and some APIs for interacting with in-memory representations of resources that use this language.

There are various concrete syntaxes that can be used to transmit resources that use this abstract language, two of which are defined in this specification.

The first such concrete syntax is the

The DOM, the HTML syntax, and the XHTML syntax cannot all represent the same content. For example, namespaces cannot be represented using the HTML syntax, but they are supported in the DOM and in the XHTML syntax. Similarly, documents that use the `<noscript>` feature can be represented using the HTML syntax, but cannot be represented with the DOM or in the XHTML syntax.

Comments that contain the string "-->" can only be represented in the DOM, not in the HTML and XHTML syntaxes.

Non-normative materials providing
a context for the HTML

§2 Common infrastructure

§3 Semantics, structure, and APIs of HTML documents

§4 The elements of HTML

§5 User interaction

§6 Loading Web pages

§7 Web application APIs

§8 The HTML syntax

§10 Rendering

There are also some appendices, listing [§11 Obsolete features](#) and [§12 IANA considerations](#), and several indices.

This specification should be read like all other specifications. First, it should be read cover-to-cover, multiple times. Then, it should be read backwards at least once. Then it should be read by picking random sections from the contents list and following all the cross-references.

As described in the conformance requirements section below, this specification describes conformance criteria for a variety of conformance classes. In particular, there are conformance requirements that apply to *producers*, for example authors and the documents they create, and there are conformance requirements that apply to *consumers*, for example Web browsers. They can be distinguished by what they are requiring: a requirement on a producer states what is allowed, while a requirement on a consumer states how software is to act.

¶

(EXAMPLE) For example, "the foo attribute's value must be a valid integer" is a requirement on producers, as it lays out the allowed values; in contrast, the requirement "the foo attribute's value must be parsed using the rules for parsing integers" is a requirement on consumers, as it describes how to process the content.

Requirements on producers have no bearing whatsoever on consumers.

¶

(EXAMPLE) Continuing the above example, a requirement stating that a particular attribute's value is constrained to being a [valid integer](#) emphatically does *not* imply anything about the requirements on consumers. It might be that the consumers are in fact required to treat the attribute as an opaque string, completely unaffected by whether the value conforms to the requirements or not. It might be (as in the previous example) that the consumers are required to parse the value using specific rules that define how invalid (non-numeric in this case) values are to be processed.

20

This is a definition, requirement, or explanation.

This is a note.

¶

(EXAMPLE) This is an example.

This is an open issue.

This is a warning.

```
interface Example {
    // this is an IDL definition
};
```

```
variable = object . method( [ optionalArgument ] )
```

This is a note to authors describing the usage of an interface.

```
/* this is a CSS fragment */
```

The defining instance of a term is marked up like **this**. Uses of that term are marked up like [this](#) or like [this](#).

The defining instance of an element, attribute, or API is marked up like `this`. References to that element, attribute, or API are marked up like `<this>`.

Other code fragments are marked up like this.

Byte sequences with bytes in the range 0x00 to 0x7F, inclusive, are marked up like this.

Variables are marked up like *this*.

In an algorithm, steps in synchronous sections are marked with ?.

In some cases, requirements are given in the form of lists with conditions and corresponding requirements. In such cases, the requirements that apply to a condition are always the first set of requirements that follow the condition, even in the case of there being multiple sets of conditions for those requirements. Such cases are presented as follows:

?This is a condition

?This is another condition

This is the requirement that applies to the conditions above.

?This is a third condition

This is the requirement that applies to the third condition.

§ 1.8. Privacy concerns

This section is non-normative.

Some features of HTML trade user convenience for a measure of user privacy.

In general, due to the Internet's architecture, a user can be distinguished from another by the user's IP address. IP addresses do not perfectly match to a user; as a user moves from device to device, or from network to network, their IP address will change; similarly, NAT routing, proxy servers, and shared computers enable packets that appear to all come from a single IP address to actually map to multiple users. Technologies such as onion routing can be used to further anonymize requests so that requests from a single user at one node on the Internet appear to come from many disparate parts of the network.

However, the IP address used for a user's requests is not the only mechanism by which a user's requests could be related to each other. Cookies, for example, are designed specifically to enable this, and are the basis of most of the Web's session features that enable you to log into a site with which you have an account.

There are other mechanisms that are more subtle. Certain characteristics of a user's system can be used to distinguish groups of users from each other; by collecting enough such information, an individual user's browser's "digital fingerprint" can be computed, which can be as good, if not better, as an IP address in ascertaining which requests are from the same user.

Grouping requests in this manner, especially across multiple sites, can be used for both benign (and even arguably positive) purposes, as well as for malevolent purposes. An example of a reasonably benign purpose would be

Other features in the platform can be

used for the same purpose, though,
including, though not limited to:

- The exact list of which features a user agents supports.
- The maximum allowed stack depth for recursion in script.
- Features that describe the user?s environment, like Media Queries and the [Screen](#) object.
[\[MEDIAQ\]](#) [\[CSSOM-VIEW\]](#)
- The user?s time zone.

§ 1.9. A quick introduction to HTML

This section is non-normative.

A basic HTML document looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sample page</title>
  >
  </head>
  <body>
    <h1>Sample page</h1>
    <p>This is a <a href=
"demo.html">simple</a>
sample.</p>
  <!-- this is a comment -->
  </body>
</html>
```

HTML documents consist of a tree of elements and text. Each element is denoted in the source by a start tag, such as "`<body>`", and an end tag, such as "`</body>`". (Certain start tags and end tags can in certain cases be omitted and are implied by other tags.)

Tags have to be nested such that elements are all completely within each other, without overlapping:

```
<p>This is <em>very <strong>
wrong</em>!</strong></p>
```



```
<p>This <em>is <strong>
correct</strong>.</em></p>
```

This specification defines a set of elements that can be used in HTML, along with rules about the ways in which the elements can be nested.

Elements can have attributes, which control how the elements work. In the example below, there is a [hyperlink](#), formed using the `<a>` element and its `href` attribute:

```
<a href="demo.html">simple</a>
```

Attributes are placed inside the start tag, and consist of a name and a value, separated by an "=" character. The attribute value can remain unquoted if it doesn't contain space characters or any of " ' ` = < or >. Otherwise, it has to be quoted using either single or double quotes. The value, along with the "=" character, can be omitted altogether if the value is the empty string.

```
<!-- empty attributes -->
<input name=address disabled>

<input name=address disabled=
">">

<!-- attributes with a value -->

<input name=address maxlength
=200>
<input name=address maxlength
='200'>
<input name=address maxlength
="200">
```

HTML user agents (e.g., Web browsers) then parse this markup, turning it into a DOM (Document Object Model) tree. A DOM tree is an in-memory representation of a document.

DOM trees contain several kinds of nodes, in particular a DocumentType node, Element nodes, Text nodes,

[Comment](#) nodes, and in some cases [ProcessingInstruction](#) nodes.

The [markup snippet at the top of this section](#) would be turned into the following DOM tree:

- DOCTYPE: html
- [<html>](#)
 - [<head>](#)
 - #text: ???
 - [<title>](#)
 - #text: Sample page
 - #text: ??
 - #text: ??
 - [<body>](#)
 - #text: ???
 - [<h1>](#)
 - #text: Sample page
 - #text: ???
 - [<p>](#)
 - #text: This is a
 - [<a>](#) href="demo.html"
 - #text: simple
 - #text: sample.
 - #text: ???
 - #comment: this is a comment
 - #text: ???

The [document](#) element of this tree is the [<html>](#) element, which is the element always found in that position in HTML documents. It contains two elements, [<head>](#) and [<body>](#), as well as a [Text](#) node between them.

There are many more [Text](#) nodes in the DOM tree than one would initially expect, because the source contains a number of spaces (represented here by "?") and line breaks ("") that all end up as [Text](#) nodes in the DOM. However, for historical reasons not all of the spaces and line breaks in the original markup appear in the DOM. In

The `<head>` element contains a `<title>` element, which itself contains a `Text` node with the text "Sample page". Similarly, the `<body>` element contains an `<h1>` element, a `<p>` element, and a comment.

```
<form name="main">
  Result: <output name=
"result"></output>
  <script>
    document.forms.main.
elements.result.value =
'Hello World';
  </script>
</form>
```

```
var a = document.links[0]; // obtain the
a.href = 'sample.html'; // change the des
a.protocol = 'https'; // change just the
a.setAttribute('href',
'http://example.com/'); // change the con
```

Since DOM trees are used as the way to represent HTML documents when

In the following example, the page has been made yellow-on-blue using CSS.

For more details on how to use HTML, authors are encouraged to consult tutorials and guides. Some of the examples included in this specification might also be of use, but the novice author is cautioned that this specification, by necessity, defines the language with a level of detail that might be difficult to understand at first.

When HTML is used to create

A comprehensive study of this matter is beyond the scope of this document, and authors are strongly encouraged to study the matter in more detail. However, this section attempts to provide a quick introduction to some common pitfalls in HTML application development.

Not validating user input
Cross-site scripting (XSS)
SQL injection

When writing filters to validate user input, it is imperative that filters always be safelist-based, allowing known-safe constructs and disallowing all other input. Blocklist-based filters that disallow known-bad inputs and allow everything else are not secure, as not everything that is bad is yet known (for example, because it might be invented in the future).

¶

(EXAMPLE) For example, suppose a page looked at its URL's query string to determine what to display, and the site then redirected the user to that page to display a message, as in:

```
<ul>
  <li><a href="message."
>Say Hello</a>
  <li><a href="message."
>Say Welcome</a>
  <li><a href="message."
>Say Kittens</a>
</ul>
```

If the message was just displayed to the user without escaping, a hostile attacker could then craft a URL that contained a script element:

http://example.com/message.cgi?say=%3C

If the attacker then convinced a victim user to visit this page, a script of the attacker's choosing would run on the page. Such a script could do any number of hostile actions, limited only by what the site offers: if the site is an e-commerce shop, for instance, such a script could cause the user to unknowingly make arbitrarily many unwanted purchases.

This is called a cross-site scripting attack.

There are many constructs that can be used to try to trick a site into executing code. Here are some that authors are encouraged to consider when writing safelist filters:

When allowing harmless-seeming elements like ``, it is important to safelist any provided attributes

as well. If one allowed all attributes then an attacker could, for instance, use the `onload` attribute to run arbitrary script.

- When allowing URLs to be provided (e.g., for links), the scheme of each URL also needs to be explicitly safelisted, as there are many schemes that can be abused. The most prominent example is "javascript:", but user agents can implement (and indeed, have historically implemented) others.
- Allowing a `<base>` element to be inserted means any `<script>` elements in the page with relative links can be hijacked, and similarly that any form submissions can get redirected to a hostile site.

Cross-site request forgery (CSRF)

If a site allows a user to make form submissions with user-specific side-effects, for example posting messages on a forum under the user's name, making purchases, or applying for a passport, it is important to verify that the request was made by the user intentionally, rather than by another site tricking the user into making the request unknowingly.

This problem exists because HTML forms can be submitted to other origins.

Sites can prevent such attacks by populating forms with user-specific hidden tokens, or by checking `Origin` headers on all requests.

Clickjacking

A page that provides users with an interface to perform actions that the user might not wish to perform needs to be designed so as to avoid the possibility that users can be tricked into activating the interface.

One way that a user could be so tricked is if a hostile site places the

To avoid this, sites that do not expect to be used in frames are encouraged to only enable their interface if they detect that they are not in a frame (e.g., by comparing the window object to the value of the `top` attribute).

This section is non-normative.

On the other hand, parsing of HTML files happens incrementally, meaning that the parser can pause at any point to let scripts run. This is generally a good thing, but it does mean that authors need to be careful to avoid hooking event handlers after the events could have possibly fired.

'EXAMPLE' COUNTER
(EXAMPLE) One way this could manifest itself is with `` elements and the `load` event. The

Here, the author uses the `onload` handler on an `` element to catch the `load` event:

If the element is being added by script, then so long as the event handlers are added in the same script, the event will still not be missed:

However, if the author first created the `` element and then in a separate script added the event listeners, there's a chance that the load event would be fired in between, leading it to be missed:

```
<!-- Do not use this style, it has a race condition -->
<img id="games" src=
"games.png" alt="Games">
<!-- the 'load' event might fire here while the
break, in which case you will not see it

<script>
var img = document.
getElementById('games');
img.onload =
    gamesLogoHasLoaded;
// might never fire!
</script>
```

This section is non-normative.

1.10. Conformance requirements for authors

This section is non-normative.

However, even though the processing of invalid content is in most cases well-defined, conformance requirements for documents are still important: in practice, interoperability (the situation in which all implementations process particular content in a reliable and identical or equivalent way) is not the only goal of document conformance requirements. This section details some of the more common reasons for still distinguishing between a [conforming document](#) and one with errors.

This section is non-normative.

The use of presentational elements leads to poorer accessibility

While it is possible to use presentational markup in a way

Using media-independent markup, on the other hand, provides an easy way for documents to be authored in such a way that they are "accessible" for more users (e.g., users of text browsers).

It is significantly easier to maintain a site written in such a way that the markup is style-independent. For example, changing the color of a site that uses `` throughout requires changes across the entire site, whereas a similar change to a site based on CSS can be done by changing a single file.

Presentational markup tends to be much more redundant, and thus results in larger document sizes.

The only remaining presentational markup features in HTML are the [style](#) attribute and the [<style>](#) element. Use of the [style](#) attribute is somewhat discouraged in production environments, but it can be useful for rapid prototyping (where its rules can

It is also worth noting that some elements that were previously presentational have been redefined in this specification to be media-independent: ``, `<i>`, `<hr>`, `<s>`, `<small>`, `<u>`.

This section is non-normative.

Unintuitive error-handling behavior

Certain invalid syntax constructs, when parsed, result in DOM trees that are highly unintuitive.

```
<table><hr>...
```

To allow user agents to be used in controlled environments without having to implement the more bizarre and convoluted error handling rules, user agents are permitted to fail whenever encountering a **parse error**.

Some error-handling behavior,

Errors that can result in info set coercion

Errors that result in disproportionately poor performance

Certain syntax constructs can result in disproportionately poor performance. To discourage the use of such constructs, they are typically made non-conforming.

१

(EXAMPLE) For example, the following markup results in poor performance, since all the unclosed `<i>` elements have to be reconstructed in each paragraph, resulting in progressively more elements in each paragraph:

```
<p><i>He dreamt.  
<p><i>He dreamt that he  
<p><i>Then lunch.  
<p><i>  
And finally dinner.
```

The resulting DOM for this fragment would be:

- <u>
 ◦ <i>
 ■ #text: He
 dreamt.
- <p>
 ◦ <i>
 ■ <i>
 ■ #text: He
 dreamt that
 he ate
 breakfast.
- <p>
 ◦ <i>
 ■ <i>
 ■ <i>
 ■ #text:
 Then
 lunch.
- <p>
 ◦ <i>
 ■ <i>
 ■ <i>
 ■ <i>
 ■ #tex
 : And
 finally
 dinner

Thus, the correct way to

```
<a href=
"?art&amp;copy">
Art and Copy</a> <!-- the & has to k
```

Certain syntax constructs are known to cause especially subtle or serious problems in legacy user agents, and are therefore marked as non-conforming to help authors avoid them.

'EXAMPLE ' COUNTER
(EXAMPLE) Another example of this is the DOCTYPE, which is required to trigger no-quirks mode, because the behavior of legacy user agents in quirks mode is often largely undocumented.

Certain restrictions exist purely to avoid known security problems.

'EXAMPLE ' COUNTER
(EXAMPLE) For example, the restriction on using UTF-7 exists purely to avoid authors falling prey to a known cross-site-scripting attack using UTF-7. [\[RFC2152\]](#)

Markup where the author's intent is very unclear is often made non-conforming. Correcting these errors early makes later maintenance easier.

Contact details

When a user makes a simple typo, it is helpful if the error can be caught early, as this can save the author a lot of debugging time. This specification therefore usually considers it an error to use element names, attribute names, and so forth, that do not match the names defined in this specification.

In order to allow the language syntax to be extended in the future, certain otherwise harmless features are disallowed.

(EXAMPLE) For example, attributes in end tags are ignored currently, but they are invalid, in case a future change to the language makes use of that syntax feature without conflicting with already-deployed (and valid!) content.

Some authors find it helpful to be in the practice of always quoting all attributes and always including all optional tags, preferring the consistency derived from such custom over the minor benefits of terseness afforded by making use of the flexibility of the HTML syntax. To aid such authors, conformance checkers can provide modes of operation wherein such conventions are enforced.

This section is non-normative.

Beyond the syntax of the language, this specification also places restrictions on how elements and attributes can be specified. These restrictions are present for similar reasons:

To avoid misuse of elements with defined meanings, content models are defined that restrict how elements can be nested when such nestings would be of dubious value.

(EXAMPLE) For example, this specification disallows nesting a `<section>` element inside a `<kbd>` element, since it is highly unlikely for an author to indicate that an entire section should be keyed in.

Errors that involve a conflict in expressed semantics

Similarly, to draw the author's attention to mistakes in the use of elements, clear contradictions in the semantics expressed are also considered conformance errors.

'EXAMPLE ' COUNTER (EXAMPLE) In the fragments below, for example, the semantics are nonsensical: a separator cannot simultaneously be a cell, nor can a radio button be a progress bar.

```
<hr role="cell">
```

```
<input type=radio  
role=progressbar>
```

'EXAMPLE ' COUNTER (EXAMPLE) Another example is the restrictions on the content models of the `` element, which only allows `` element children. Lists by definition consist just of zero or more list items, so if a `` element contains something other than an `` element, it's not clear what was meant.

Cases where the default styles are likely to lead to confusion

Certain elements have default styles or behaviors that make certain combinations likely to lead to confusion. Where these have equivalent alternatives without this problem, the confusing combinations are disallowed.

'EXAMPLE' COUNTER (EXAMPLE) Another example would be the way interactive content cannot be nested. For example, a `<button>` element cannot contain a `<textarea>` element. This is because the default behavior of such nesting interactive elements would be highly confusing to users. Instead of nesting these elements, they can be placed side by side.

Sometimes, something is disallowed because allowing it would likely cause author confusion.

(EXAMPLE) For example, setting the `disabled` attribute to the value "false" is disallowed, because despite the appearance of meaning that the element is enabled, it in fact means that the element is *disabled* (what matters for implementations is the presence of the attribute, not its value).

Some conformance errors simplify the language that authors need to learn.

(EXAMPLE) For example, the `<area>` element's `shape` attribute, despite accepting both `"circ"` and `"circle"` values in practice as synonyms, disallows the use of the `"circ"` value, so as to simplify tutorials and other learning aids. There would be no benefit to allowing both, but it would cause extra confusion when teaching the language.

Certain elements are parsed in somewhat eccentric ways (typically for historical reasons), and their content model restrictions are intended to avoid exposing the author to these issues.

```
<p>Welcome. </p><form>
<label>Name:</label>
  <input></form>
```

Some errors are intended to help prevent script problems that would be hard to debug.

(EXAMPLE) This is why, for instance, it is non-conforming to have two `id` attributes with the same value. Duplicate IDs lead to the wrong element being selected, with sometimes disastrous effects whose cause is hard to determine.

Some constructs are disallowed because historically they have been the cause of a lot of wasted authoring time, and by encouraging authors to avoid making them, authors can save time in future efforts.

(EXAMPLE) For example, a `<script>` element's `src` attribute causes the element's contents to be ignored. However, this isn't obvious, especially if the element's contents appear to be executable script ? which can lead to authors spending a lot of time trying to debug the inline script without realizing that it is not executing. To reduce this problem, this specification makes it non-conforming to have executable script in a `<script>` element when the `src` attribute is present. This means that authors who are validating their documents are less likely to waste time with this kind of mistake.

Some authors like to write files that can be interpreted as both XML and HTML with similar results. Though this practice is discouraged in general due to the myriad of subtle complications involved (especially when involving scripting, styling, or any kind of automated serialization), this specification has a few restrictions intended to at least somewhat mitigate the difficulties. This makes it easier for authors to use this as a transitional step when migrating between HTML and XHTML.

(EXAMPLE) For example, there are somewhat complicated rules surrounding the lang and xml:lang attributes intended to keep the two synchronized.

(EXAMPLE) Another example would be the restrictions on the values of `xmlns` attributes in the HTML serialization, which are intended to ensure that elements in conforming documents end up in the same namespaces whether processed as HTML or XML.

As with the restrictions on the syntax intended to allow for new syntax in future revisions of the language, some restrictions on the content models of elements and values of attributes are intended to allow for future expansion of the HTML vocabulary.

(EXAMPLE) For example, limiting the values of the target attribute that start with an U+005F LOW LINE character (_) to only specific predefined values allows new predefined values to be introduced at a future time without conflicting with author-defined values.

Certain restrictions are intended to support the restrictions made by other specifications.

(EXAMPLE) For example, requiring that attributes that take media query lists use only *valid* media query lists reinforces the importance of following the conformance rules of that specification.

§ 1.11. Suggested reading

This section is non-normative.

The following documents might be of interest to readers of this specification.

Character Model for the World Wide Web 1.0: Fundamentals [CHARMOD]

This Architectural Specification provides authors of specifications, software developers, and content developers with a common reference for interoperable text manipulation on the World Wide Web, building on the Universal Character Set, defined jointly by the Unicode specification and ISO/IEC 10646. Topics addressed include use of the terms "character", "encoding" and "string", a reference processing model, choice and identification of character encodings, character escaping, and string indexing.

Unicode Security Considerations

Web Content Accessibility Guidelines (WCAG) 2.0 [WCAG20]

Authoring Tool Accessibility Guidelines (ATAG) 2.0 [ATAG20]

User Agent Accessibility Guidelines (UAAG) 2.0 [UAAG20]

HTML Accessibility APIs Mappings 1.0 [html-aam-1.0]

