that namespace? make it equivalent to editing the first text node. But doing it as textContent is the simplest from a specification point of view. Opinions?

Web Forms 2.0

Working Draft 9 March 2004

This version:

http://www.hixie.ch/specs/html/forms/web-forms-3

Latest version:

http://www.hixie.ch/specs/html/forms/web-forms

Previous versions:

http://www.hixie.ch/specs/html/forms/web-forms-2
http://www.hixie.ch/specs/html/forms/xforms-basic-1
http://lists.w3.org/Archives/Member/w3c-archive/2003Sep/att-0014/hfp.html

Editor:

Ian Hickson, Opera Software, ian@hixie.ch

Abstract

This specification defines Web Forms 2.0, an extension to the forms features found in HTML 4.01's forms chapter. Web Forms 2.0 applies to both HTML and XHTML user agents, and provides new strongly-typed input fields, new attributes for defining constraints, a repeating model for declarative repeating of form sections, new DOM interfaces, new DOM events for validation and dependency tracking, and XML submission and initialization of forms. This specification also standardises and codifies existing practice in areas that have not been previously documented.

HTML4, XHTML1.1 and the DOM are thus extended in a manner which has a clear migration path from existing HTML forms, leveraging the knowledge

authors have built up with their experience with HTML so far.

Status of this document

This is a work in progress! This document is changing on a daily if not hourly basis in response to comments and as a general part of its development process. Comments are very welcome, please send them to htmlforms@damowmow.com and cc www-archive@w3.org. Thank you.

It is very wrong to cite this as anything other than a work in progress. Do not implement this in a production product. It is not ready yet! At all!

This document currently has no official standing within the W3C at all. It is the result of loose collaboration between interested parties over dinner, in various mailing lists, on IRC, and in private e-mail. To become involved in the development of this document, please send comments to the address given above. **Your input will be taken into consideration.**

This is a working draft and may therefore be updated, replaced or rendered obsolete by other documents at any time. It is inappropriate to use Working Drafts as reference material or to cite them as other than "work in progress".

This draft contains a couple of namespaces that use the `data:` URI scheme. These are temporary and **will be changed** before this specification is ready to be implemented.

To find the latest version of this working draft, please follow the "Latest version" link above.

Table of contents

1. Introduction
    1.1. Relationship to HTML
    1.2. Relationship to XHTML
    1.3. Relationship to XForms
    1.4. Relationship to XForms Basic
    1.5. Missing features
    1.6. Conformance requirements

1.7. Terminology
2. Extensions to form control elements
2.1. Extensions to the `input` element
2.1.1. Ranges
2.1.2. Precision
2.2. The `output` element
2.3. Extensions to the `textarea` element
2.4. Extensions to file upload controls
2.5. Extensions to the `form` element
2.6. Extensions to the submit buttons
2.7. Extensions to existing attributes
2.8. The `pattern` attribute
2.9. The `required` attribute
2.10. The `form` attribute
2.11. The `autocomplete` attribute
2.12. The `inputmode` attribute
2.13. The `help` attribute
2.14. Handling unexpected elements and values
3. Repeating form controls
3.1. Introduction for authors
3.1.1. What the repetition model can't do
3.2. Definitions
3.2.1. Repetition templates
3.2.2. Repetition blocks
3.3. New form controls
3.4. Event interface for repetition events
3.5. The repetition model
3.5.1. Addition
3.5.2. Removal
3.5.3. Movement of repetition blocks
3.5.4. Initial repetition blocks
3.6. Examples
3.6.1. Repeated rows
3.6.2. Nested repeats
4. The forms event model
4.1. Scope resolution for ECMAScript in HTML event handler attributes
4.2. Change events and input events
4.3. Events to enable simpler dependency tracking
4.4. Form validation
4.5. Receiving the results of form submission
5. Form submission
5.1. Successful form controls
5.2. Handling characters outside the submission character set

5.3. `application/x-www-form-urlencoded`
5.4. `application/x-www-form+xml`: XML submission
5.5. `text/plain`
5.6. Semantics of `method` and `enctype` attributes
5.6.1. For `http:` actions
5.6.2. For `ftp:` actions
5.6.3. For `data:` actions
5.6.4. For `file:` actions
5.6.5. For `mailto:` actions
5.6.6. For `smsto:` and `sms:` actions
5.6.7. For `javascript:` actions
6. Fetching data from external resources
6.1. Filling `select` elements
6.2. Seeding a form with initial values
7. Extensions to the HTML Level 2 DOM interfaces
7.1. Additions specific to the `HTMLFormElement` interface
7.2. Additions specific to the `HTMLSelectElement` interface
7.2.1. The `HTMLCollection` interface
7.3. The `HTMLOutputElement` interface
7.4. Validation APIs
7.5. New DOM attributes for new content attributes
7.6. Labels
7.7. Repetition interfaces
7.8. Loading remote documents
8. Styling form controls
8.1. Relation to the CSS3 User Interface module
A. XHTML module definition
B. Attribute summary
References
Acknowledgements

---

1. Introduction

This is an update to the forms features found in HTML 4.01's forms chapter, which are informally referred to as Web Forms 1.0.

Authors have long requested changes to HTML4 to support some of their more common needs. For example, take this extract from a recent post written by an anonymous poster on the popular topic-driven Slashdot forum:

> *There are three things that need adjustments to get decent forms in HTML.*

> *First, have the option of not redrawing the page upon submission. [...] Second, have a "grid" widget that allows spreadsheet-like data entry grids.*
>
> *Third, have validation options such as <input type="text" name="foo" format="number" decimals=2> or perhaps <input type="number" name="foo" decimals=2>*

This post is typical of the kind of comments made by Web authors. Requirements from such comments in mailing lists and other discussions have been examined and from these sources a set of requirements and design goals were derived:

- Backwards compatibility (where possible).
- Ease of authoring for authors with limited knowledge about XML, data models, etc, but familiar with commonly used languages such as HTML and ECMAScript.
- Basic data typing, providing new controls for commonly used types, so that authors do not need to repeatedly design complicated widgets such as calendars.
- Simpler validation on the client side (while recognizing that server side validation will still be required), with declarative solutions for the common case but strong DOM support so that less common cases can easily be handled using scripting.
- Dynamically adding more fields (repeating structures) on the client side without scripting.
- XML submission (although not necessarily structured XML submission).
- The ability to initialize forms from external data sources, so that authors do not have to dynamically rewrite the form content itself to prefill forms, but can instead use static pages with scripts that dynamically generate only the data part.
- Few dependencies: This specification should stand alone and not require support for other technologies such as XPath, XMLSchema, or XML Events, as these are not implemented in typical Web browsers and requiring them would significantly delay the time required to deploy browsers supporting this specification.
- This specification should be implementable in full on devices with limited resources.

Not all the desired features have been included in this specification. Future

versions may be introduced to address further neads.

This specification does not describe the complete behaviour of an HTML or XHTML user agent. Readers are expected to refer to the existing specifications for the definitions of features that this specification does not change.

1.1. Relationship to HTML

This specification clarifies and extends the semantics put forth in [HTML4] for form controls and form submission. It is expected to be implemented in ordinary HTML user agents alongside existing forms technology, and indeed, some of the features described in this draft have been implemented by user agents as ad-hoc, non-standard extensions for many years due to strong market need.

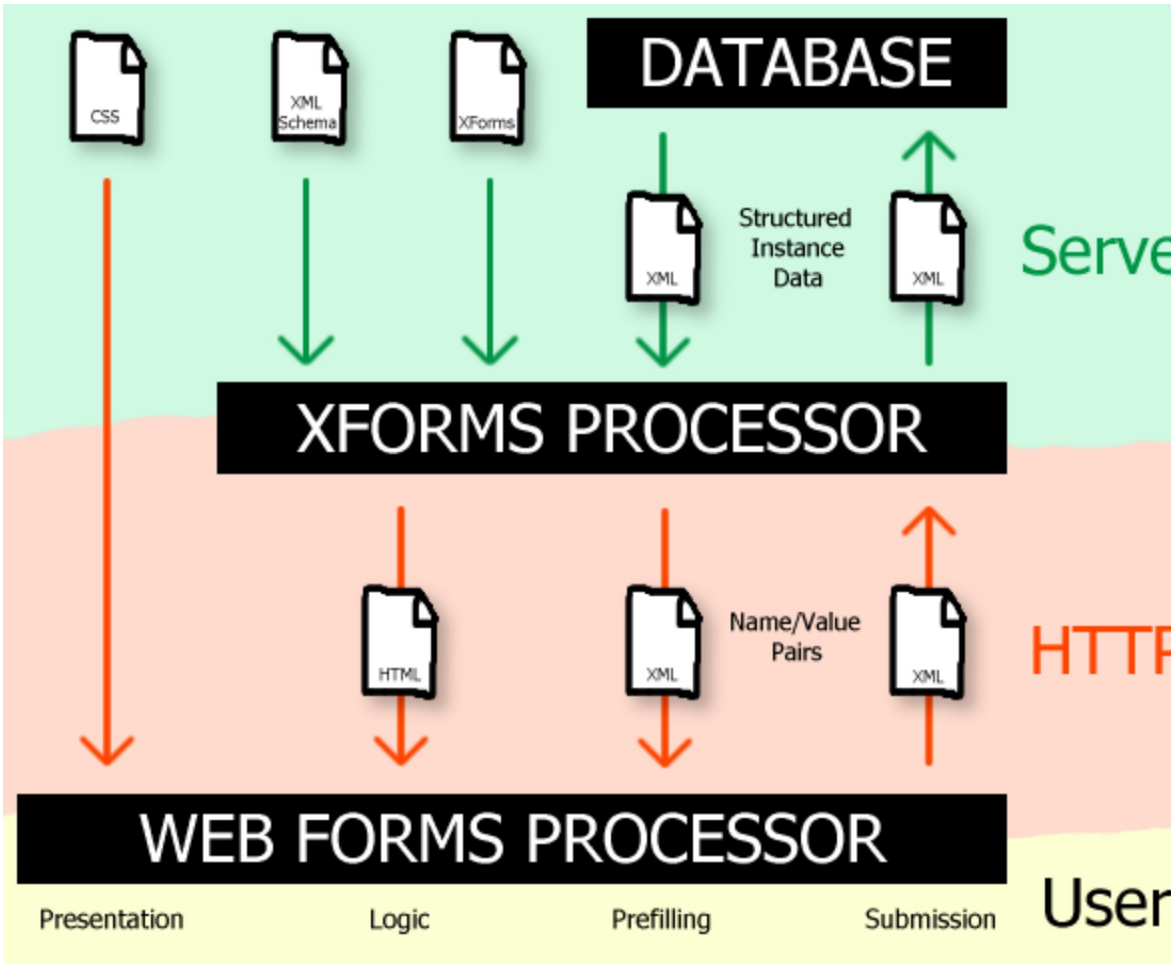
1.2. Relationship to XHTML

This specification can also be viewed as an extension to [XHTML1]. In particular, some of the features added in this module only apply to XHTML documents, for example features allowing mixed namespaces.

1.3. Relationship to XForms

This specification is in no way aimed at replacing XForms 1.0 [XForms], nor is it a subset of XForms 1.0.

XForms 1.0 is well suited for describing business logic and data constraints. Unfortunately, due to its requirements on technologies not widely supported by Web browsers, it has not been widely implemented by those browsers itself. This specification aims to simplify the task of transforming XForms 1.0 systems into documents that can be rendered on every day Web browsers.



In this transformation model, the XForms processor is a server-side process that converts XForms and XML Schema documents, according to the XForms specification, into HTML and Web Forms documents, which are then processed by the client side Web Forms processor, along with a style sheet for presentation.

The structured XML instance data stored on the server-side (e.g. in a database) is converted by the XForms processor into name/value pairs that are then used by the UA to prefill the form. Submission follows the opposite path, with the UA generating name/value pairs and sending them to the XForms processor on the server, which converts them back into structured XML for storage or further processing.

In order to simplify this transformation process, this specification attempts to add some of the functionality of XForms with a minimum impact on the existing, widely implemented forms model. Where appropriate, backwards compatibility, ease of authoring, and ease of implementation have been given priority over theoretical purity.

The following features of XForms have *not* been addressed:

- The separation of the instance data model, data typing, field

interdependencies, and submission information from the content model and interface elements.

- The ability to create arbitrary XML fragments to be filled in before submission.
- The ability to edit local XML files directly. (While technically not defined by the XForms 1.0 specification, UAs have generally implemented such a feature since it is easy to extend the XForms model in that way.)
- Compound data type definitions (schemas).

The majority of the features that XForms supports using declarative syntax are, in this specification, handled by using scripting. Some new interfaces are introduced to simplify some of the more tedious tasks.

1.4. Relationship to XForms Basic

This specification is unrelated to the XForms Basic profile.

> ***Note: A previous version of this draft was called "XForms Basic". This name has been changed so as to avoid confusion with the similarly named draft from the W3C.***

1.5. Missing features

This draft does not address all needs. In addition to the features of XForms that have not been addressed (see above), the following features were considered but rejected for this version of the specification:

- Digital signatures for submissions. This is currently not covered by this specification due to patent concerns. However, it would still be considered for future inclusion if suggestions of how to support it without infringing on known patents were provided.
- DOM interfaces for the creation of new controls that are still able to interact with form submission. This need will be addressed in a separate specification.
- Elements or properties to create a "tabbed" or "wizard" interface. This need will be addressed in a separate specification.
- A rich text editing or HTML editing control. This need may be addressed in a future specification.
- A grid or spreadsheet editing control. This need may addressed in a future

version of this specification.

- A declarative way of specifying that one list should filter the view of a second list. Again, however, this need may addressed in a future version of this specification.

1.6. Conformance requirements

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Diagrams, examples, and notes are non-normative. All other content in this specification is intended to be normative.

This specification includes by reference the form-related parts of the HTML4, XHTML1.1, DOM2 HTML, DOM3 Core, and DOM3 Events specifications ([HTML4], [XHTML1], [DOM2HTML], [DOM3CORE], [DOM3EVENTS]). Compliant UAs must implement all the semantics of those specifications to claim compliance to this one.

Documents that use the new features described in this specification using HTML over HTTP must be served as `text/html`.

Documents that use the new features described in this specification using XHTML or other XML languages over HTTP must be served using an XML MIME type such as `application/xml` or `application/xhtml+xml`. [RFC3023]

This specification introduces attributes for setting the maximum size or range of certain values. While user agents should support all possible values, there may be implementation specific limits.

1.7. Terminology

This specification refers to both HTML and XML attributes and DOM attributes, often in the same context. When it is not clear which is being referred to, they are referred to as **content attributes** for HTML and XML attributes, and **DOM attributes** for those from the DOM. Similarly, the term "properties" is used for both ECMAScript object properties and CSS properties. When these are ambiguous they are simply qualified as object properties and CSS properties respectively.

Generally, when the specification states that a feature applies to HTML or XHTML, it also includes the other. When a feature specifically only applies to one of the two languages, it is called out explicitly, as in:

> *...it is possible that authors would prefer to declare the page's forms in advance, in the* `head` *element of XHTML documents (this does not apply to HTML documents).*

Unless otherwise stated, XML elements defined in this specification are elements in the `http://www.w3.org/1999/xhtml/` namespace, and attributes defined in this specification have no namespace. This does not apply to HTML as HTML does not support namespaces.

2. Extensions to form control elements

HTML `input` elements use the `type` attribute to specify the data type. In [HTML4], the types (as seen by the server) are as follows:

`text`

A free-form text input, nominally free of line breaks.

`password`

A free-form text input for sensitive information, nominally free of line breaks.

`checkbox`

A set of zero or more values from a predefined list that is expected to be structurally separated from its related values (in the limiting case of the list only containing one value, this is equivalent to a boolean).

`radio`

An enumerated value that is expected to be structurally separated from its related values.

`submit`

An enumerated value, with the extra semantic that it must be the last value selected and initiates form submission.

`file`

An arbitrary file with a MIME type and optionally a file name.

`image`

A coordinate, relative to a particular image's size, with the extra semantic that it must be the last value selected and initiates form submission.

`hidden`

An arbitrary string that is not made available to the user.

In addition, HTML also provides a few alternate elements that convey typing semantics similar to the above types, but use different content models:

`select`

An enumerated value, whose values are structurally kept together.

`select multiple`

A set of zero or more values from a predefined list, much like the `checkbox` type, whose values are structurally kept together.

`textarea`

A free-form text input, nominally with no line break restrictions.

`button`

An enumerated value, with the extra semantic that it must be the last value selected and initiates form submission, much like the `submit` type but with a richer content model.

There are also two button types (available on both `input` and `button` elements) that are never submitted: `button` and `reset`.

This specification includes all of these types, their semantics, and their processing rules, by reference, for backwards compatibility. Compliant UAs must follow all the guidelines given in the HTML4 specification except those modified by this specification.

These types are useful, but limited. This section expands the list to cover more specific data types, and introduces attributes that are designed to constrain data entry or other aspects of the UA's behaviour.

In addition to the attributes described below, some changes are made to the content model of HTML form elements to take into account scripting needs. Specifically, the `form`, `legend`, `select`, and `optgroup` elements may now be empty (in HTML4, those elements always required at least one element child, or, in the case of `legend`, at least one character of text). The `optgroup` element may now be nested, as suggested by the HTML4 specification.

Also, as controls no longer need to be contained within their `form` element to be associated with it, it is possible that authors would prefer to declare the page's forms in advance, in the `head` element of XHTML documents (this does not apply to HTML documents). This is therefore allowed, although only when the `form` element is empty.

Similarly, `form` elements in XHTML may now be nested (this does not apply to HTML). Form controls by default associate with their nearest form ancestor. Forms are not related to ancestor forms in any way semantically, and do not

share attributes or form controls or events (except insofar as events bubble up the DOM).

The `form` and `select` elements are extended with `data` attributes for fetching values and options from external resources.

2.1. Extensions to the `input` element

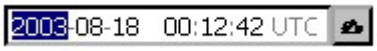
Several new types are introduced for the `type` attribute. As with the older types, UAs are recommended to show specialized widgets for these types, instead of requiring that the user enter the data into a text field.

The formats described below are those that UAs must use when submitting the data. They do not necessarily represent what the user is expected to type. It is the UA's responsibility to convert the user's input into the specified format.

`datetime`

A date and time (year, month, day, hour, minute, second) encoded according to [ISO8601] with the time zone set to UTC, e.g.: `1995-12-31T23:59:59Z`. User agents are expected to show an appropriate widget.

Note: This specification does not specify how the widget should appear. It could be something like this:



UAs may display the time in whatever time zone is appropriate for the user, but should be clear to the user that the time is globally defined, not time-zone dependent. The submitted date and time must be in the UTC timezone.

`date`

A date (year, month, day) encoded according to [ISO8601], e.g.: `1995-12-31`. User agents are expected to show an appropriate widget, such as a calendar.

`expdate`

A date consisting of a year and a month encoded according to [ISO8601], e.g.: `1995-12`. This type is used most frequently for credit card expiry dates.

`week`

A date consisting of a year and a week number encoded according to [ISO8601], e.g.: `1996-W52`. This type is used most frequently for dates in European industry.

time

A time (hour, minute) encoded according to [ISO8601] with no time zone, e.g.: `23:59`. User agents are expected to show an appropriate widget, such as a clock. UAs should make it clear to the user that the time does not carry any time zone information.

number

A number. The allowed precision of the number decides what UI user agents may show, and is dicussed below, under the `precision` attribute.

Numbers must be submitted as follows: an optional minus sign ("-"), one or more decimal integers, optionally a decimal point (".") and a decimal fractional part, together forming a number representing the base, followed optionally by the lowercase literal letter "e", another optional minus sign, and a decimal integer exponent representing the index of a power of ten with which to multiply the base to get the resulting number. If the exponent part is omitted it must be assumed to be zero.

For example, negative-root-two, to 32 significant figures, would be `-1.41421356237309504880168872420 97e0`, the radius of the earth given in furlongs would be `3.17053408e4`, and the answer to the life, the universe and everything could be any of (amongst others) `42`, `0042.000`, `42e0`, `4.2e1`, or `420e-1`.

This format is designed to be compatible with `scanf(3)`'s `%f` format, ECMAScript's `parseFloat`, and similar parsers while being easier to parse than required by some other floating point syntaxes.

Note that `+0`, `0e+0`, `+0e0` are invalid numbers (the minus sign cannot be replaced by a plus sign for positive numbers, it must simply be dropped). UAs must not submit numbers in invalid formats.

The submission format is not intended to be the format seen and used by users. UAs may use whatever format and UI is appropriate for user interaction; the description above is simply the submission format.

range

Same as `number`, but indicates that the exact value is not important, letting UAs optimise their UI for usability. For instance, if this is specified with `min` and `max` attributes, visual UAs may use a track bar control. The `precision` attribute still applies.

email

An e-mail address, as defined by [RFC2822] (the `addr-spec` token, defined in RFC2822 section 3.4.1, excluding the `CFWS` subtoken everywhere and

the `FWS` subtoken everywhere except in the `quoted-string` subtoken). UAs could, for example, offer e-mail addresses from the user's address book.

`tel`

A telephone number, as defined by [RFC2806] (the `global-phone-number` token, defined in RFC2806 section 2.2).

`uri`

A URI, as defined by [RFC2396] (the `absoluteURI` token, defined in RFC2396 section 3). UAs could, for example, offer the user URIs from his bookmarks.

`location`

A geographical coordinate, specified as two floating point numbers (an optional negative sign, one or more decimal digits, a decimal point, and six more decimal digits) separated by a comma. The value specifies latitude and longitude, in that order, as decimal degrees. The latitude represents the location north and south of the equator as a positive or negative real number, respectively, in the range -90.000000 ≤ θ ≤ 90.000000. The longitude represents the location east and west of the prime meridian as a positive or negative real number, respectively, in the range -180.000000 < φ ≤ 180.000000. The longitude and latitude values must be specified as decimal degrees and must be specified to six decimal places. This allows for granularity within a meter of the geographical position.
Servers should ignore data following a second comma (in other words, the data is really a comma separated list, and currently only the first two fields are defined). This will allow for future extension of this field. Clients should only specify coordinates that are accurate to at least a few hundred meters. User agents may offer "bookmarked" locations for the user's convenience, or offer a map-based control for coordinate selection, or offer the current location as determined by GPS, or use other interfaces. User agents should not automatically send the user's location without the user's consent.

> For example, the value `37.386013,-122.082932` is a coordinate near Santa Cruz, in California, USA.

Empty fields (those with no value) do not need to match their type. (Although if they are required fields, they will stop submission for that reason anyway.)

On the other hand, fields that are not successful (such as disabled controls) do not take part in submission, and therefore are simply not checked for validity.

> The following form uses some of the types described above:
>
> ```
> <form action="..." method="post" onsubmit="verify(event)">
> ```

```
<p>
 <label>
  Quantity:
  <input name="count" type="number" min="0" max="99" value="1" />
 </label>
</p>
<p>
 <label for="time1"> Preferred delivery time: </label>
 <input id="time1" name="time1" type="time" min="08:00:00" max="17
 <input id="time2" name="time2" type="time" min="08:00:00" max="17
</p>
<script type="text/javascript">
 function verify(event) {
   // check that time1 is smaller than time2, otherwise, swap them
   if (event.target.time1.value &gt;= event.target.time2.value) {
     var time2Value = event.target.time2.value;
     event.target.time2.value = event.target.time1.value;
     event.target.time1.value = time2Value;
   }
 }
</script>
</form>
```

Servers should still perform type checking on submitted data, as malicious users or rogue user agents might submit data intended to bypass this client-side type checking. Validation done via script may also be easily bypassed if the user has disabled scripting.

The `size` attribute of the `input` element is deprecated in favor of using CSS to specify the layout of the form.

2.1.1. Ranges

To limit the range of values allowed by the above types, two new attributes are introduced, which apply to the date-related, time-related, numeric, and file upload types:

`min`

Gives the minimum value (inclusive) of the field, in the format specified for the relevant type. Values for the field less than the minimum value are out of range (ERROR_RANGE_UNDERFLOW). If absent, or if the minimum value is not in exactly the expected format, there is no minimum restriction.

`max`

Gives the maximum value (inclusive) of the field, in the format specified for the relevant type. Values for the field greater than the maximum value are out of range (ERROR_RANGE_OVERFLOW). If absent, or if the maximum value is not in exactly the expected format, there is no

maximum restriction.

For date, time and numeric fields, the values indicate the allowed range. For file upload fields, the values indicate the allowed number of files.

The ERROR_TYPE_MISMATCH code is used for fields whose values do not match their types, and the ERROR_RANGE_UNDERFLOW and ERROR_RANGE_OVERFLOW codes are used for fields whose values are outside the allowed range.

A field with a `max` less than its `min` can never be satisfied and thus would block a form from being submitted. This is not not make the document non-conformant.

2.1.2. Precision

An extra attribute is also introduced to control the precision for the `number` and `range` type:

precision

This attribute specifies the maximum allowed precision of the number. Precision must be given in one of the following forms:

***n*dp**

A specified number of decimal places. *n* must be an integer (one or more digits in the range 0-9). This specifies how many digits may come after the decimal point when the number is serialised with a zero exponent, ignoring trailing zeros. Zero itself is always valid (assuming it is within the range of `min` and `max` of course).

> These numbers have precisions of no more than 2dp: 0, 1, 1.23, 0.123e1, 123e-1, 123456789.010

> These numbers have precisions of more than 2dp: 0.001, 1.234, 0.1234e1, 123e-3, 123456789.001

***n*sf**

A specified number of decimal significant figures. *n* must be an integer (one or more digits in the range 0-9). This specifies how many digits may come after the decimal point when the number is serialised with an exponent such that the integer part is zero, and the first decimal is non-zero, ignoring trailing zeros. Zero itself is always valid (assuming it is within the range of `min` and `max` of course).

> These numbers have precisions of no more than 2sf: 0, 1, 1.20, 0.120e1, 120e-1, 120000000.000

> These numbers have precisions of more than 2sf: 0.00123,

> 123, 1.23, 0.123e1, 123e-1, 123000000.000

integer
: **The default.** Same as `0dp`. If the field doesn't match any of the other values, it should be treated as this value.

float
: No precision restrictions.

The ERROR_PRECISION_EXCEEDED code is used for fields whose numbers have more precision than allowed by the `precision` attribute. However, UAs may silently round the number to the maximum precision instead of reporting a validation error.

User agents are recommended to never convert user- and author-supplied values to their binary numeric representation, keeping the values in string form at all times and performing comparisons in that form. This ensures that UAs are able to handle arbitrarily large numbers without risking data loss due to rounding in the decimal-to-binary conversion.

If a UA needs to round a number to its nearest binary equivalent, for example when converting a user-supplied decimal number and an author-supplied minimum in order to compare them to establish validity (ignoring the suggestion above to do these comparisons in string form), algorithms equivalent to those specified in ECMA262 sections 9.3.1 ("ToNumber Applied to the String Type") and 8.5 ("The Number type") should be used (possibly after suitably altering the algorithms to handle numbers of the range that the UA can support). [ECMA262]

2.2. The `output` element

The `output` element acts very much like a `span` element, except that it is considered to be a form control for the purposes of the DOM. Its namespace is the same as for the other form control elements, `http://www.w3.org/1999/xhtml`. It has no attributes beyond the common attributes and the `form` attribute. Its value is given by its contents, which must be only text (like the `textarea` element). Its value can be set dynamically via the `value` DOM attribute, thus replacing the contents of the element.

The *initial value* of the `output` control is stored in a mutable `defaultValue` DOM attribute of type `DOMString`. This is similar to the way `textarea` elements work, except that the contents of an element for `output` controls reflects the *current value* not the initial, or default, value. See [HTML4] section 17.2 for the definiton of the term "initial value".

The `output` element is never successful for form submission. Resetting a form

does reset its `output` elements.

> The following example shows two input fields. Changing either field updates an `output` element containing the product of both fields.
>
> ```
> <form>
>  
>   <input name="a" type="number" precision="float" value="0"> *
>   <input name="b" type="number" precision="float" value="0"> =
>   <output name="result" onforminput="value = a.value * b.value">0</
>  
> </form>
> ```
>
> This would work something like the following:
>
> 0 * 0 = 0
>
> ***Note: The `forminput` event is defined in the section on new events.***

2.3. Extensions to the `textarea` element

The `rows` and `cols` attributes of the `textarea` element are no longer required attributes. When unspecified, CSS-compliant browsers should lay the element out as specified by CSS, and non-CSS UAs may use UA-specific defaults, such as, for visual UAs, using the width of the display device and a height suitable for the device.

The `textarea` element may have a **`wrap`** attribute specified. This attribute controls the wrapping behaviour of submitted text.

`soft`

> This is the default value. The text is submitted without line breaks other than explicitly entered line breaks. (In other words, the submitted text is exactly as found in the DOM.)

`hard`

> The text is submitted with explicit line breaks, and in addition, line breaks added to wrap the text at the width given by the `cols` attribute. (These additional line breaks can't be seen in the DOM.)

Authors should always specify a `cols` attribute when the `wrap` attribute is set to `hard`. When `wrap="hard"` is specified without a `cols` attribute, user agents should use the display width when wrapping the text for submission. This will typically mean that different users submit text at different wrapping widths, defeating much of the purpose of client-side wrapping.

CSS UAs should *render* `textarea` elements as specified by the `'white-space'` property, although UAs may have rules in their UA stylesheet that key the default 'white-space' property values based on the `wrap` element for `textarea` elements.

The `maxlength` attribute applies to `textarea` controls.

2.4. Extensions to file upload controls

File upload controls (`input` elements of type **`file`**) are not successful if the user enters a value that specifies non-existent files. There is no error code for this situation because that would open the way for some privacy or security leaks. It is recommended that user agents report problems of this nature to the user.

The `min` and `max` attributes apply to file upload controls and specify (as positive integers) how many files must be attached for the control to be valid. They default to 0 and 1 respectively (and so limit the default number of files to 1 optional file, as per most existing implementatios in early 2004). The ERROR_RANGE_UNDERFLOW and ERROR_RANGE_OVERFLOW codes are used to indicate when fields do not have the specified number of files selected.

The **`accept`** attribute may be used to specify a comma-separated list of content types that a server processing the form will handle correctly. This attribute was specified in [HTML4]. In this specification, this attribute is extended as follows:

- MIME types may have a subtype of `*`, for example:

  ```
  <input type="file" name="avatar" accept="image/*"/>
  ```

  In this way, the `accept` attribute may be used to specify that the server is expecting an image, a sound clip, a video, etc, without specifying the exact list of types.

- UAs should use the list of acceptable types in constructing a filter for a file picker, if one is provided to the user.

- If the UA wishes to let the user create the file prior to upload, it should use the `accept` attribute's MIME type list to determine which application to use.

  > One recent use for sound file upload has been the concept of *audio blogging*. This is similar to straight-forward Web logging, or diary writing, but instead of submitting textual entries, one submits sound bites.
  >
  > The submission interface to such a system could be written as follows:

```
<form action="/weblog/submit" method="post" enctype="multipart
<label>
Attach your audio-blog sound file:
<input type="file" name="blog" accept="audio/*"/>
</label>
<input type="submit" value="Blog!"/>
</form>
```

A compliant UA could, upon encountering this form, provide a "Record" button instead of, or in addition to, the more usual "Browse" button. Selecting this button could then bring up a sound recording application.

This is expected to be most useful on small devices that do not have file systems and for which the only way of handling file upload is to generate the content on the fly.

- The ERROR_TYPE_MISMATCH code is used to indicate that at least one of the selected files does not have a MIME type conforming to one of the MIME types listed as acceptable. UAs may allow the user to override the MIME type to be one of the allowable types if the file is originally incorrectly labeled (but should not allow users to override the type merely to let submission continue, as that would defeat the point of having a restriction in the first place).

- If an `accept` attribute is set on a `form` element, it sets the default for any file upload controls in that form. (This is done by the file upload controls first checking their attribute, and if they don't have one, checking their form's. The two attributes don't "stack".)

The `maxlength` attribute applies to file upload controls.

2.5. Extensions to the `form` element

The `form` element's `action` attribute is no longer a required attribute. If omitted, the default value is the empty string, which is a relative URI pointing at the current document (or the specified base URI, if any).

To support incremental updates of forms, a new attribute is introduced on the `form` element: **`replace`**. This attribute takes two values:

document

The default value. The entire document (as specified by the `target` attribute when the document contains frames) is replaced by the return value.

values

The body returned from the server is treated as a new data file for prefilling the form.

Note: These names, and their exact semantics, differ from those of the equivalent attribute in XForms 1.0 (the `replace` attribute on the `submission` element). The equivalent of this specification's `document` is equivalent to the XForms `all`, and the equivalent of `values` is `instance`. The equivalent of the XForms `none` value is `document` with the server returning an HTTP 204 No Content return code.

The exact semantics are described in detail in the section on submission, under step nine.

2.6. Extensions to the submit buttons

Normally, activating a submit button (an `input` or `button` element with the `type` attribute set to `submit`, or an `input` element with the `type` attribute set to `image`) submits the form, using the form's submission details (`action`, `method`, `enctype`, and `replace` attributes).

In some cases, authors would like to be able to submit a form to different processors, using different submission methods, or not replacing the form but just updating the details with new data. For this reason, the following attributes are allowed on submit buttons: `action`, `method`, `enctype`, `replace`, and `target`. When not specified, their values default to the values given by their `form` element.

If a submit button is activated, then the submission uses the values as given by the button that caused the activation, with missing attributes having their values taken from the form.

2.7. Extensions to existing attributes

In addition to the new attributes given in this section, some existing attributes from [HTML4] are clarified and extended below. These, and other attributes from HTML4, continue having the same semantics as described in HTML4 unless specified otherwise.

`disabled`

The `disabled` attribute applies to all control types, including `fieldset` (in HTML4 the `disabled` attribute did not apply to the `fieldset` element), except the `output` element.

`maxlength`

This attribute applies to `text`, `password` and `file` input types, and `textarea` elements. In particular, it does not apply to the date-related, time-related, and numeric field types, or to the `email`, `tel`, or `uri` types. ***In HTML4, this attribute only applied to the `text` and `password` types.***

For text input controls it specifies the maximum length of the input, in terms of numbers of characters. For details on counting string lengths, see [CHARMOD].

When specified on a file upload control, it specifies the maximum size in bytes of the content.

The ERROR_TOO_LONG code is used when this attribute is specified on a `text`, `password`, or `textarea` control and the control has more than the specified number of characters, or when it is specifies on a `file` control and at least one of the selected files is longer than the specified number of bytes.

Servers should still expect to receive, and must be able to cope with, content larger than allowed by the maxlength attribute, in order to deal with malicious or non-conforming clients.

`name`

Some names (all starting with the string "`Ecom_`") in this version of HTML forms have predefined meanings, allowing UAs to fill in the form fields automatically. These names, and their semantics, are described in [RFC3106].

`readonly`

This attribute applies only to `text`, `password`, `email`, `tel`, `uri`, date-related, time-related, and numeric input types, as well as the `textarea` element. Specifically, it does not apply to radio buttons, check boxes, file upload fields, `select` elements, or any of the button types; the interface concept of "readonly" values does not apply to button-like interfaces. (The DOM `readonly` attribute ([DOM2HTML]) obviously applies to the same set of types as the HTML attribute.)

Other attributes not listed in this specification retain the same semantics as in [HTML4].

2.8. The `pattern` attribute

For the `text`, `email`, `tel`, and `uri` types of the `input` element and the `textarea` element, a new attribute, **`pattern`**, is introduced to specify patterns that the strings must match.

When specified, the `pattern` attribute contains a regular expression that the field's value must match before the form may be submitted (ERROR_PATTERN_MISMATCH).

```
<label> Credit Card Number:
 <input type="text" pattern="^[0-9]{10}$" name="cc" />
</label>
```

The regular expression language used for this attribute is the same as that defined in [ECMA262], except that the `pattern` attribute implies a `^` at the start of the pattern and a `$` at the end (so the pattern must match the entire value, not just any subset). If the attribute is empty or omitted then it is equivalent to `.*` (which, with the implied start and end characters, becomes `^.*$`), which matches anything.

In the case of the `email`, `tel`, and `uri`, the `pattern` attribute specifies a pattern that must be matched *in addition* to the value matching the generic pattern relevant for the field. If the pattern given by the attribute specifies a pattern that is incompatible with the grammar of the field type, as in the example below, then the field could never be satisfied. (A document containing such a situation is not technically invalid, but it is of dubious semantic use.)

```
<form>
 <p>
  This form could never be submitted, as the following required field
  can never be satisfied:
  <input type="uri" pattern="^[^:]+$" required="required" name="test"/>
 </p>
</form>
```

When the value doesn't match the field's type, a ERROR_TYPE_MISMATCH error occurs; when the value doesn't match the pattern, a ERROR_PATTERN_MISMATCH error occurs.

2.9. The `required` attribute

Form controls can have the **`required`** attribute specified, to indicate that the user must enter a value into the form control before submitting the form.

The `required` attribute applies to all form controls except check boxes, radio buttons, controls with the type `hidden`, image inputs, buttons, `fieldset`s, and `output` elements. It *can* be used on controls with the `readonly` attribute set; this may be useful in scripted environments. For disabled controls, the attribute has no effect.

The ERROR_REQUIRED code is used for form controls marked as required that do not have values.

> Here is a form fragment showing two required fields and one optional field. A user agent would not allow the user to submit the form until the "name" and "team" fields were filled in.
>
> ```
> <ul>
>   <li>Name: <input type="text" name="name" required="required" /></
>   <li>Team:
>     <select name="team" required="required">
>       <option value="foxes">The Foxes</option>
>       <option value="ferrets">The Ferrets</option>
>       <option value="kittens">The Kittens</option>
>     </select>
>   <li>Comment: <input type="text" name="comment" /></li>
>   </li>
> </ul>
> ```

Any non-empty value satisfies the `required` condition, including a simple whitespace character.

2.10. The `form` attribute

All form controls can have the `form` attribute specified. The `form` attribute gives the ID of the `form` element the form control should be associated with, and overrides the relationship between the form control and any ancestor `form` element.

Setting an element's `form` attribute either to a non-existent ID, to the empty string, or to an ID that identifies an element that is not an HTML `form` element, disassociates the form control from its form, leaving it unassociated with any form.

When set on a `fieldset` element, this also changes the association of any descendant form controls, unless they have `form` attributes of their own, or are contained inside forms that are themselves descendants of the `fieldset` element.

When forms are submitted, reset, or have their form controls enumerated through the DOM, only those controls associated with the form are taken into account. A control can be associated only with one form at a time.

A `form` attribute that specifies an ID that occurs multiple times in a document should select the same form as would be selected by the `getElementById()` method for that ID ([DOM3CORE]).

> In this example, each row contains one form, even though without this attribute it would not be possible to have more than one form per table if any of them span cells.

```
<table>
 <thead>
  <tr>
   <th>Name</th>
   <th>Value</th>
   <th>Action</th>
  </tr>
 </thead>
 <tbody>
  <tr>
   <td>
    <form id="edit1" action="/edit" method="post">
     <input type="hidden" name="id" value="1"/>
     <input type="text" name="name" value="First Row"/>
    </form>
   </td>
   <td>
    <input form="edit1" type="text" name="value"/>
   </td>
   <td>
    <input form="edit1" type="submit" name="Edit"/>
   </td>
  </tr>
  <tr>
   <td>
    <form id="edit2" action="/edit" method="post">
     <input type="hidden" name="id" value="2"/>
     <input type="text" name="name" value="Second Row"/>
    </form>
   </td>
   <td>
    <input form="edit2" type="text" name="value"/>
   </td>
   <td>
    <input form="edit2" type="submit" name="Edit"/>
   </td>
  </tr>
 </tbody>
</table>
```

2.11. The `autocomplete` attribute

All form controls except the various push button controls and `hidden` and `output` controls, can have the **`autocomplete`** attribute set. The attribute takes two values, `on` and `off`. The default, when the attribute is not specified, is `on`.

The on value means the UA is allowed to store the value entered by the user so that if the user returns to the page, the UA can pre-fill the form. The off value means that the UA must not remember that field's value.

> Banks frequently do not want UAs to pre-fill login information:

```
<p>Account: <input type="text" name="ac" autocomplete="off" /></li>
<p>PIN: <input type="text" name="pin" autocomplete="off" /></li>
```

Note: In practice, this attribute is required by many banking institutions, who insist that UAs implement it before supporting them on their Web sites. For this reason, it is implemented by most major Web browsers already, and has been for many years.

2.12. The `inputmode` attribute

The **`inputmode`** attribute applies to the `input` element when it has a `type` attribute of `text`, `password`, `email`, `tel`, or `uri`, and to the `textarea` element.

This attribute is defined to be exactly equivalent to the `inputmode` attribute defined in the XForms 1.0 specification (sections E1 through E3.2) [XForms].

2.13. The `help` attribute

Any form control can have a `help` attribute specified. This attribute contains a URI that the UA may use to provide help information regarding the active field.

This specification does not specify how help information should be used, but for example, the UA could show a small pop-up window if the user focuses such a control and pressed the `F1` key, or could show the help information in a side-bar while the relevant control is focused.

Note: This attribute is added mainly because XForms has it, to show that it would be trivial to add to HTML as well. However, there is some doubt that it is actually a useful feature. The XForms `hint` element is already supported in HTML, as the `title` attribute.

2.14. Handling unexpected elements and values

There are several elements that are defined as expecting particular elements as children. Using the DOM, or in XML, it is possible for authors to violate these expectations and place elements in unexpected places.

Authors must not do this. User agent implementors may curse authors who violate these rules, and may persecute them to the full extent allowed by applicable international law.

Upon encountering such an invalid construct, UAs must proceed as follows:

For parsing errors in HTML

This document does not specify exact parsing semantics for ambiguous cases that are not covered by SGML. UA implementors should devine appropriate behaviour by reverse engineering existing products and attempting to emulate their behaviour. (This does not apply to XHTML, since the XML specification specifies mandatory formal error handling rules.)

For non-empty `form` elements in `head` elements in XHTML

Typically UAs are expected to hide all the contents of `head` elements. No other special behaviour is required to cope with this case; if the author overrides this hiding (e.g. through CSS) then the form must behave like any other form. (This does not apply to HTML, where a `form` in a `head` would, per SGML parsing rules, imply a `body` start tag.)

For non-empty `input` elements

By default, the form control must replace the contents of the element in the rendering with the form control widget. Using CSS3 Generated Content [CSS3CONTENT] or XBL [XBL], however, it is possible for the author to override this behaviour.

For `output` elements containing elements

The `defaultValue` DOM attribute is initialized from the DOM3 Core `textContent` attribute ([DOM3CORE]). Setting the element's `value` attribute is defined to be identical to setting the DOM3 Core `textContent` attribute. While the element contains elements, they are rendered according to the CSS rules.

For `textarea` elements containing elements

The `defaultValue` DOM attribute is identical to the `textContent` DOM attribute both for reading and writing, and is used to set the initial `value`. The rendering is based on the `value` DOM attribute, not the contents of the element, unless CSS is used to override this somehow.

For `select` elements containing nodes other than `option` and `optgroup` elements, and for `optgroup` elements containing nodes other than `option` elements

Only the `option` and `optgroup` elements take part in the `select` semantics. Unless otherwise forced to appear by a stylesheet, other child nodes are never visible.

For `option` elements containing nodes other than text nodes

The value of the control, if not specified explicitly, is initialized using the `textContent` DOM attribute's value.

As far as rendering goes, it is left largely up to the UA. Two possibilities are sensible: rendering the content normally, just as it would have been

outside the form control; and rendering the initial value only, with the rest of the content not displayed (unless forced to appear through some CSS).

Note: It should be noted that while nesting a form inside a `select` control may look cool, it is extremely poor UI and must not be encouraged.

For `option` and `optgroup` elements that are not inside `select` elements

The elements should be treated much like `span` elements as far as rendering goes.

For attributes that contain invalid values

The attribute must be ignored. It will appear in the DOM, but not affect the form semantics. For example, if a `min` attribute on a `datetime` control is an integer instead of a date and time string, then the range has no minimum. If the `type` attribute is then changed to `number`, then the attribute would take effect.

For labels pointing (via `for`) to elements that are not form controls

The attribute must be ignored. It will appear in the DOM (including as the value of `htmlFor`) but the `control` DOM attribute must return null and activating the label must not send focus to the associated element.

For `repeat` elements with children or attributes other than the `index` attribute

Children and attributes are automatically ignored since inserting the element into the document results in its immediate removal.

Other invalid cases should be handled analogously.

3. Repeating form controls

Occasionally forms contain repeating sections, for example an order form could have one row per item, with product, quantity, and subtotal fields. The **repeating form controls model** defines how such a form can be described without resorting to scripting.

Note: The entire model can be emulated purely using JavaScript and the DOM. With such a library, this model could be used and down-level clients could be supported before user agents implemented it ubiquitously. Creating such a library is left as an exercise to the reader.

3.1. Introduction for authors

This subsection is not normative.

Occasionally, a form may have a section to be repeated an arbitrary number of times. For example, an order form could have one row per item. Traditionally, this has been implemented either by using complex client-side scripts or by sending a request to the server for every new row.

Using the mechanisms described in this section, the problem is reduced to describing a template in the markup, and then specifying where and when that template should be repeated.

To explain this, we will step through an example. Here is a sample form with three rows:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
 <head>
  <title>Sample Order Form</title>
 </head>
 <body>
  <table>
   <tr>
    <th>Product</th>
    <th>Quantity</th>
   </tr>
   <tr>
    <td><input type="text" name="row0.product" value=""></td>
    <td><input type="text" name="row0.quantity" value="1"></td>
   </tr>
   <tr>
    <td><input type="text" name="row1.product" value=""></td>
    <td><input type="text" name="row1.quantity" value="1"></td>
   </tr>
   <tr>
    <td><input type="text" name="row2.product" value=""></td>
    <td><input type="text" name="row2.quantity" value="1"></td>
   </tr>
  </table>
 </body>
</html>
```

The template for those rows could look something like:

```
<tr>
 <td><input type="text" name="row0.product" value=""></td>
 <td><input type="text" name="row0.quantity" value="1"></td>
</tr>
```

...except that then the names would all be the same — all rows would be

"row0", so there would be no clear way of distinguishing which "quantity" went with which "product" except by the order in which they were submitted.

To get around this, the template is modified slightly:

```
<tr id="order">
 <td><input type="text" name="row[order].product" value=""></td>
 <td><input type="text" name="row[order].quantity" value="1"></td>
</tr>
```

The template now has a unique identifier, and that identifier is used to indicate where the row index should be substituted in. When a template is replicated, all the attributes containing the template's id between square bracket characters (`[id]`) have that ID replaced by a unique index.

In order to distinguish this row from a normal row, however, something needs to be added to the template to mark it as being a template. This is done using a `repeat` attribute:

```
<tr id="order" repeat="template">
 <td><input type="text" name="row[order].product" value=""></td>
 <td><input type="text" name="row[order].quantity" value="1"></td>
</tr>
```

If we replace the table with that markup:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
 <head>
  <title>Sample Order Form</title>
 </head>
 <body>
  <table>
   <tr>
    <th>Product</th>
    <th>Quantity</th>
   </tr>
   <tr id="order" repeat="template">
    <td><input type="text" name="row[order].product" value=""></td>
    <td><input type="text" name="row[order].quantity" value="1"></td>
   </tr>
  </table>
 </body>
</html>
```

...then nothing but the header will appear! This is because templates are not rendered. Templates have to be *repeated*. This is done with the `repeat` element:

```
...
<tr id="order" repeat="template">
 <td><input type="text" name="row[order].product" value=""></td>
 <td><input type="text" name="row[order].quantity" value="1"></td>
```

```
    </tr>
    <repeat>
    <repeat>
    <repeat>
   </table>
  </body>
 </html>
```

This is now identical to the original example. It still isn't dynamic — there is no way for the user to add more rows.

This can be solved by adding an `add` button. The `add` button type adds a copy of a template when the user presses the button, in much the same way as the `repeat` element does.

There are two ways to use `add` buttons. The first is by explicitly specifying which template should be replicated:

```
<p><input type="add" template="order" value="Add Row"></p>
```

The template is specified using a `template` attribute on the `input type="add"` or `button type="add"` element. The `template` attribute contains an ID that should match the ID of the template you want the button to affect.

When such a button is pressed, the template is replicated, and the resulting block is inserted just after the last block that is associated with the template. For example, there are three rows in the example above, so if the user pressed that button, the new block would be inserted just after the third one.

The second way is by including an `add` button inside the template, so that when the template is replicated, the button is replicated into the resulting block. When such a button is pressed, the template is replicated, and inserted immediately before the block in which the button is found. For example, if there were `add` buttons in the rows of the example above, and someone pressed the button in the second row, a row would be inserted between the first row and the second row.

For this example we will only use the first way:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
 <head>
  <title>Sample Order Form</title>
 </head>
 <body>
  <table>
   <tr>
    <th>Product</th>
    <th>Quantity</th>
   </tr>
```

```
   <tr id="order" repeat="template">
    <td><input type="text" name="row[order].product" value=""></td>
    <td><input type="text" name="row[order].quantity" value="1"></td>
   </tr>
   <repeat>
   <repeat>
   <repeat>
  </table>
  <p><input type="add" template="order" value="Add Row"></p>
 </body>
</html>
```

Now the user can add more rows, but he cannot remove them. Removing rows is done via the `remove` button type. When a user presses such a button, the row in which the button is kept is removed from the document.

```
<input type="remove" value="Remove This Row">
```

This is added to the template so that it appears on every row:

```
   <tr id="order" repeat="template">
    <td><input type="text" name="row[order].product" value=""></td>
    <td><input type="text" name="row[order].quantity" value="1"></td>
    <td><input type="remove" value="Remove This Row"></td>
   </tr>
```

The final result looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
 <head>
  <title>Sample Order Form</title>
 </head>
 <body>
  <table>
   <tr>
    <th>Product</th>
    <th>Quantity</th>
   </tr>
   <tr id="order" repeat="template">
    <td><input type="text" name="row[order].product" value=""></td>
    <td><input type="text" name="row[order].quantity" value="1"></td>
    <td><input type="remove" value="Remove This Row"></td>
   </tr>
   <repeat>
   <repeat>
   <repeat>
  </table>
  <p><input type="add" template="order" value="Add Row"></p>
  <p><input type="submit" value="Submit"></p>
 </body>
</html>
```

If the user pressed "Add" once, removed the middle two rows, typed in some garbage in the two "product" text fields, and pressed "Submit", the user agent would submit the following name-value pairs:

```
order0.product=some
order0.quantity=1
order3.product=garbage
order3.quantity=1
```

Further examples are given in the examples section below.

The repetition model supports more than just the cases given above, for instance there are `move-up` and `move-down` buttons that can be inserted inside templates much like the `remove` button but for moving rows up and down.

Repetition templates can also be nested. The concept of hierarchy is expected to be represented in the names, as it is today in hand-rolled repeating forms, as in:

```
order1.name
order1.quantity
order1.comment1.text
order1.comment2.text
order2.name
order2.quantity
order2.comment1.text
```

That way the submission can remain compatible with the long-established `multipart/form-data`, yet not lose the structure of the data.

> ***Note: The naming schemes used above are arbitrary. Any naming scheme could be used, at the convenience of the author.***

3.1.1. What the repetition model can't do

The current repetition model can't declaratively limit the number of repeats (although you can write script to do this manually). This specification also does not address the ability to select a template to move it up or down without using buttons directly associated with the current template.

3.2. Definitions

> ***Note: In this section, a number of references are made to namespaces. For authors who are only using HTML or XHTML, the definitions below ensure that no namespaces need appear in the document (except the namespace on the root element). Thus, such a reader can simply gloss over the parts that mention***

namespaces.

In order to implement such a form declaratively, a new global attribute is introduced: the **`repeat` attribute**. When placed on elements in the `http://www.w3.org/1999/xhtml` namespace, it must be a namespace-free attribute, and when placed on other elements, it must be an attribute in the `http://www.w3.org/1999/xhtml` namespace.

The effect of this attribute depends on its value, which can be either the literal string "`template`", or an integer.

3.2.1. Repetition templates

An element in the `http://www.w3.org/1999/xhtml` namespace with the `repeat` attribute in no namespace, or an element in any other namespace with the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace, with the attribute's value equal to `template`, is a **repetition template**.

Repetition templates may occur anywhere. They are not specifically associated with any form.

Every template has an index associated with it. The initial value of a template's index is always 0. The index is used to ensure that when cloning templates, the new block has a unique ID. The template's index does not appear in the markup. (It does, however, appear in the DOM, as the `repetitionIndex` attribute.)

Unrecognized tokens must be ignored.

```
<div repeat="template"/> <!-- A template. -->
<div repeat="template +1 3"/> <!-- Not a template. -->
<div repeat=" template"/> <!-- Not a template (leading whitespace). -->
```

3.2.2. Repetition blocks

An element in the `http://www.w3.org/1999/xhtml` namespace with the `repeat` attribute in no namespace, or an element in any other namespace with the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace, with the attribute's value equal to an integer (an optional leading '-' character followed by one or more decimal digits), is a **repetition block**.

Repetition blocks should only occur as following siblings of repetition templates. If an element is declared as a repetition block but does not have a previous sibling that is a repetition template, then it can only take part in certain aspects of the repetition model (namely deletion and movement, and not addition). Such elements are termed **orphan repetition blocks**.

Every repetition block has an index associated with it. The index's initial value is the value of the `repeat` attribute.

```
<div>
 <div repeat="template"/> <!-- The template for the next few elements.
 <div repeat="0"/> <!-- A simple repetition block, index 0. -->
 <div repeat="-5"/> <!-- Another, index -5 -->
 <div repeat="2"/> <!-- A simple repetition block, index 2. -->
 <div repeat="nothing"/> <!-- Just a normal element. -->
 <div repeat=" 3"/> <!-- Another normal element (leading whitespace). -
</div>
<div repeat="0"/> <!-- Orphan repetition block, index 0. -->
```

3.3. New form controls

Several new button types are introduced to support the repetition model. These values are valid types for both the `input` element and the `button` element.

`add`

Adds a new repetition block.

`remove`

Removes the nearest ancestor repetition block.

`move-up`

Moves the nearest ancestor repetition block up one.

`move-down`

Moves the nearest ancestor repetition block down one.

These control types can never be successful.

Invoking these buttons generates events (for instance `click`), as specified by the DOM specifications. The default action for these events is to act as described below. However, if the event is cancelled, then the default action will not occur.

In addition, to support the `add` type, a new attribute is introduced to the `input` and `button` elements: `template`.

`template`

Specifies the repetition template to use.

These are described in more detail in the next section.

3.4. Event interface for repetition events

The repetition model includes several events. These use the following interface

to store their context information.

```
/* Similar to the UIEvent interface */
interface RepetitionEvent : Event {
  readonly attribute RepetitionElement element;
  void               initRepetitionEvent(in DOMString typeArg,
                                         in boolean canBubbleArg,
                                         in boolean cancelableArg,
                                         in RepetitionElement elementAr
  void               initRepetitionEventNS(in DOMString namespaceURIArg
                                           in DOMString typeArg,
                                           in boolean canBubbleArg,
                                           in boolean cancelableArg,
                                           in RepetitionElement element
};
```

The `initRepetitionEvent()` and `initRepetitionEventNS()` methods have the same behaviours as the `initEvent()` and `initEventNS()` events from [DOM3EVENTS].

3.5. The repetition model

A repetition template should not be displayed. In CSS-aware user agents, this should be achieved by including the following rules, or their equivalent, in the UA's user agent stylesheet:

```
@namespace html url(http://www.w3.org/1999/xhtml);
:not(html|*)[html|repeat="template"],
html|*[|repeat="template"] { display: none; }
```

Any form controls inside a repetition template are associated with their form's `templateElements` DOM attribute, and are *not* present in the form's `elements` DOM attribute, unless the relevant form is inside the template itself. Since controls in the `templateElements` attribute cannot be successful, controls inside repetition templates that would be part of forms outside the template can never be submitted and cannot be pre-filled directly when the form is pre-seeded. However, see the section on seeding a form with initial values for details on how repetition blocks can be pre-filled.

3.5.1. Addition

If an `add` button is activated, and it has a `template` attribute, and the element, in the same document, with the ID given by the `template` attribute in question, is a repetition template as defined above, then that element's template replication behaviour is invoked. (Specifically, in scripting-aware environments, the element's `addRepetitionBlock()` method is called with a null argument.)

If an `add` button is activated, and it has no `template` attribute, but the element

has an ancestor that is a repetition block that is not an orphan repetition block, then the repetition template associated with that repetition block has its template replication behaviour invoked with the respective repetition block as its argument. (Specifically, in scripting-aware environments, the element's `addRepetitionBlock()` method is called with a reference to the DOM Element node that represents the repetition block.)

When a template's replication behaviour is invoked (specifically, when either its **`addRepetitionBlock()`** method is called or its **`addRepetitionBlockByIndex()`** method is called) the following is performed:

1. The template examines its following siblings, up to the next repetition template or the end of the block, whichever comes first. For each sibling that is a repetition block (as defined above), if the repetition block's index is greater than or equal to the template's index, then the template's index is increased to the repetition block's index plus one. The last repetition block examined will be used in a later step.

2. If this algorithm was invoked via the `addRepetitionBlockByIndex()` method, and the value of the method's index argument is greater than the template's index, then the template's index is set to the value of index argument.

3. A clone of the template is made. The resulting element is the new repetition block element.

4. If this algorithm was invoked via the `addRepetitionBlockByIndex()` method, the new repetition block element's index is set to the method's index argument. Otherwise, the new repetition block element's index is set to the template's index.

5. If the new repetition block element is in the `http://www.w3.org/1999/xhtml` namespace, then the `repeat` attribute in no namespace on the cloned element has its value changed to the new block's index. Otherwise, the `repeat` attribute in the `http://www.w3.org/1999/xhtml` namespace has its value changed to the new block's index.

6. If the new repetition block has an ID attribute (that is, an attribute specifying an ID, regardless of the attribute's namespace or name), then that attribute's value is used as the template name in the following steps. Otherwise, the template has no name. (If there is more than one ID attribute, the "first" one in terms of node order is used. [DOM3CORE])

7. If the template has a name, then, for every attribute on the new element, and for every attribute in every descendant of the new element, any occurrences of a string consisting of an open square bracket, the

template's name, and an closing square bracket, is replaced by the new repetition block's index. (For example if the template is called "`order`", and the new repetition block's index has the value 2, and one of the attributes of one of the descendents of the new repetition block is "`order.[order].comment.[comment[order]]`", then the attribute's value is changed to "`order.2.comment.[comment2]`".) This is performed without paying attention to the types of attributes, and is done to *all* descendants, even those inside nested forms, nested repetition templates, and so forth.

8. The attribute from which the template's name was derived, if any, is removed from the new repetition block element.

9. If the first argument to the method was null, or if the argument to the function does not designate a repetition block belonging to this repetition template, then the new element is inserted into the parent of the template, immediately after the last repetition block found in the first step above, or after the template itself if there were no such repetition blocks. Mutation events are fired if appropriate.

10. Otherwise, the new element is inserted into the parent of the template, immediately *before* the node passed as the method's argument. Mutation events are fired if appropriate.

11. The template's index is increased by one.

12. An `added` event in the `data:,repetition` namespace, which bubbles but is not cancellable and has no default action, is fired on the repetition template with the repetition block's DOM node as the context information.

For an example, see the example section below.

3.5.2. Removal

If a `remove` button is activated, and the element has an ancestor that is a repetition block as defined above, then the nearest such ancestor's template deletion behaviour is invoked. (Specifically, in scripting-aware environments, the element's `removeRepetitionBlock()` method is invoked.)

When a repetition block's deletion behaviour is invoked (specifically, when its **`removeRepetitionBlock()`** method is called) the following is performed:

1. The node is removed from its parent, if it has one. Mutation events are fired if appropriate.

2. A `removed` event in the `data:,repetition` namespace, which bubbles but is not cancellable and has no default action, is fired on the element's

repetition template, if it has one, with the repetition block's DOM node as the context information.

This occurs even if the repetition block is an orphan repetition block (although if is, the event is not fired).

For an example, see the example section below.

3.5.3. Movement of repetition blocks

The two remaining button types, `move-up` and `move-down`, are used to move the current repetition block up or down the sibling repetition blocks.

If a `move-up` or `move-down` button is activated, and the element has an ancestor that is a repetition block as defined above, then the nearest such ancestor's template movement behaviour is invoked in the relevant direction. (Specifically, in scripting-aware environments, the element's `moveRepetitionBlock()` method is called; for `move-up` buttons the argument is -1 and for `move-down` buttons the argument is 1).

When a repetition block's movement behaviour is invoked (specifically, when its **`moveRepetitionBlock()`** method is called) the following is performed, where *distance* is an integer representing how far and in what direction to move the block (the argument to the method):

1. If *distance* is 0, or if the repetition block has no parent, nothing happens and the algorithm ends here.

2. Set *target*, a reference to a DOM Node, to the repetition block being moved.

3. If *distance* is negative: While *distance* is not zero and *target*'s `previousSibling` is defined and is not a repetition template, set *target* to this `previousSibling` and, if it is a repetition block, increase *distance* by one (make it less negative by one).

4. Otherwise, *distance* is positive: While *distance* is not zero and *target*'s `nextSibling` is defined and is not a repetition template, set *target* to this `nextSibling` and, if it is a repetition block, decrease *distance* by one. After the loop, set *target* to *target*'s `nextSibling` (which may be null).

5. Call the repetition block's parent node's `insertBefore()` method with the `newChild` argument being the repetition block and the `refChild` argument being *target* (which may be null by this point). Mutation events are fired if appropriate.

6. A `moved` event in the `data:,repetition` namespace, which bubbles but is

not cancellable and has no default action, is fired on the element's repetition template (if it has one) with the repetition block's DOM node as the context information.

This occurs even if the repetition block is an orphan repetition block (although if is, the event is not fired).

Moving repetition blocks does not change the index of the repetition blocks.

In addition, user agents must automatically disable `move-up` buttons (irrespective of the value of the `disabled` DOM attribute) when their repetition block could not be moved any higher according to the algorithm above, and when the buttons are not in a repetition block. Similarly, user agents must automatically disable `move-down` buttons when their repetition block could not be moved any lower according to the algorithm above, and when the buttons are not in a repetition block. This automatic disabling does not affect the DOM `disabled` property. It is an intrinsic property of these buttons.

3.5.4. Initial repetition blocks

The **`repeat` element** in the `http://www.w3.org/1999/xhtml` namespace is used to insert repetition blocks without having to explicitly copy the repetition template markup in the source document.

Authors can specify the index of the new repetition block by using the `index` attribute.

Upon being inserted into a document, `repeat` elements are immediately replaced by a repetition block created by the appropriate repetition template. The exact set of events that UAs must implement is as follows:

1. The `repeat` element is inserted into the document, either via script or during parsing. Mutation events are fired if appropriate.

2. The element's previous siblings are immediately searched, starting from the immediate previous sibling, until a repetition template is found or there are no more previous siblings.

3. If a repetition template was found and the `repeat` element has an `index` attribute, and the attribute is an integer (an optional leading '-' character followed by one or more decimal digits) then the template's `addRepetitionBlockByIndex()` method is invoked, with a reference to the `repeat` element as the first argument, and the value of the `index` attribute as the second argument.

4. Otherwise, if a repetition template was found, the template's

addRepetitionBlock() method is invoked, with a reference to the repeat element as the argument.

5. The repeat element is removed from its parent node. Mutation events are fired if appropriate.

The repeat element in HTML is an EMPTY element, so it has no end tag.

The next section shows an example.

3.6. Examples

This section gives some more practical examples of repetition.

3.6.1. Repeated rows

The following example shows how to use repetition templates to dynamically add more rows to a form in a table.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
 <head>
  <title>Form Repeat Demo</title>
 </head>
 <body>
  <form action="http://software.hixie.ch/utilities/cgi/test-tools/echo"
   <table>
    <thead>
     <tr>
      <th>Name</th>
      <th>Number of Cats</th>
      <th></th>
     </tr>
    </thead>
    <tbody>
     <tr repeat="template" id="row">
      <td><input type="text" name="name_[row]" value=""></td>
      <td><input type="text" name="count_[row]" value="1"></td>
      <td><input type="remove" value="Delete Row"></td>
     </tr>
     <tr repeat="repeated">
      <td><input type="text" name="name_0" value="John Smith"></td>
      <td><input type="text" name="count_0" value="2"></td>
      <td><input type="remove" value="Delete Row"></td>
     </tr>
     <repeat>
    </tbody>
   </table>
   <p>
    <input type="add" value="Add Row" template="row">
    <input type="submit">
```

```
   </p>
  </form>
 </body>
</html>
```

Initially, two rows would be visible, each with two text input fields, the first row having the values "John Smith" and "2", the second row having the values "" (a blank text field) and "1". The second row is the result of the `repeat` element being replaced by a repetition block while the document was being loaded.

If the "Add Row" button is pressed, a new row is added. The first such row would have the index 2 (since there are already two repetition blocks numbered 0 and 1) and so the controls would be named "name_2" and "count_2" respectively.

If the "Delete Row" button above is pressed, the row would be removed.

3.6.2. Nested repeats

The previous example does not demonstrate nested repeat blocks, reordering repetition blocks, and inserting new repetition blocks in the middle of the existing sequence, all of which are possible using the facilities described above.

This example shows nested repeats.

```
<html xmlns="http://www.w3.org/1999/xhtml">
 <head>
  <title>Solar System</title>
 </head>
 <body>
  <form>
   <h1> Solar system </h1>
   <p> <label> System Name: <input name="name"/> </label> </p>
   <h2> Planets </h2>
   <ol>
    <li repeat="template" id="planets">
     <label> Name: <input name="planet[planets].name" required="required
     <h3> Moons </h3>
     <ul>
      <li repeat="template" id="planet[planets].moons">
       <input name="planet[planets].moon[planet[planets].moons]"/>
       <input type="remove" value="Delete Moon"/>
      </li>
     </ul>
     <p><input type="add" template="planet[planets].moons" value="Add Mo
     <p><input type="remove" value="Delete Planet"/></p>
    </li>
   </ol>
   <p><input type="add" template="planets" value="Add Planet"/></p>
   <p><input type="submit"/></p>
  </form>
```

```
 </body>
</html>
```

Note that to uniquely identify each nested repeat (which is required since the `add` buttons are dependent on IDs to specify which template should have a block added), the IDs of the nested templates are specified in terms of the ancestor template's ID, using the index substitution feature.

Note: Since square brackets are not allowed in ID attributes in XML, the above example cannot validate. It is still well-formed, however.

4. The forms event model

The following events are considered form events:

- {"http://www.w3.org/2001/xml-events", "change"}
- {"http://www.w3.org/2001/xml-events", "formchange"}
- {"http://www.w3.org/2001/xml-events", "input"}
- {"http://www.w3.org/2001/xml-events", "forminput"}
- {"http://www.w3.org/2001/xml-events", "invalid"}
- {"http://www.w3.org/2001/xml-events", "submit"}
- {"http://www.w3.org/2001/xml-events", "reset"}
- {"http://www.w3.org/2001/xml-events", "received"}

Some of the above are mainly described in [DOM3EVENTS] and [HTML4]. This section introduces the new events and new semantics for the existing events.

4.1. Scope resolution for ECMAScript in HTML event handler attributes

The scope chain for ECMAScript executed in HTML event handler attributes links from the activation object for the handler, to its `this` parameter (the event target), to the form, to the document, to the default view (the window).

The event handler is passed one argument, *event*, corresponding to the event object.

Note: This definition is intentionally backwards compatible with DOM Level 0. See also ECMA-262 Edition 3, sections 10.1.6 and 10.2.3, for more details on activation objects. [ECMA262]

4.2. Change events and input events

In [DOM3EVENTS] and [HTML4], the `change` event is fired on a form control element when the control loses the input focus and its value has been modified since gaining focus.

To address the need for more immediate feedback mechanisms, this specification introduces the **`input` event**. This event is fired on a control whenever the value of the control changes due to input from the user, and is otherwise identical to the `change` event. (For example, it bubbles, is not cancelable, and has no context information.)

Change and input events must never be triggered by scripted changes to the control value. Thus, loops caused by change event handlers triggering changes are not usually possible.

Any element that accepts an `onchange` attribute to handle `change` events also accepts an `oninput` attribute to handle `input` events.

4.3. Events to enable simpler dependency tracking

Sometimes form controls are inter-dependent. In these cases, it is more intuitive to specify the dependencies on the control whose value or attributes depend on another's, rather than specify which controls should be affected by a change on the element that changes. For this reason, two new events are introduced, **`formchange`** and **`forminput`**.

These events are in the same namespace as the other form events, do not bubble, cannot be canceled, have no context information, and have no default action.

The default action of a `change` event is to fire a `formchange` event at each element in the form's `elements`, in document order, and finally at the form itself. Note that template controls are not affected. If authors need this event to affect template controls, they should hook into the form's `onformchange` event handler.

The `input` element analogously invokes the `forminput` event as its default action.

When a form is reset, a `formchange` event is fired on all the form controls of the form.

4.4. Form validation

With the introduction of the various type checking mechanisms, some way for scripting authors to hook into the type checking process is required. This is provided by the new **`invalid`** event (in the `http://www.w3.org/2001/xml-events` namespace).

When a form is submitted, each successful control in that form, in document order, is checked for validity. Then, once all the controls have been checked and the list of invalid controls, and how they are invalid, has been established, an `invalid` event must be fired on the control for each control that fails to comply with its constraints and is still a member of the form when the event is to be fired (i.e. each control whose `validity` attribute is non-zero at the start of this process).

> This definition implies a defined behaviour in the face of event handlers that mutate the document. For example, if one control's `oninvalid` attribute changes a later control's value from invalid to valid, the event is still fired on that later control. Controls added to the form during the process will not have any events fired, even if their value is invalid. Controls invalid at the start of the process that are removed from the form before receiving their events simply don't receive the event. Controls that change from one invalid state to another invalid state before receiving their event receive an event that describes their state at the *start* of the process before any events were fired.

The event can also be fired if the `validate()` method of a form control is invoked via script.

The `oninvalid` attribute (on `input`, `textarea` and `select` elements) can be used to write handlers for this event.

This event bubbles and is cancelable. The default action depends on when the event was fired.

If it was fired during form submission, then the default action is UA-specific, but is expected to consist of focusing the element (possibly firing focus events if appropriate), alerting the user that the entered value is unacceptable in the user's native language along with explanatory text saying *why* the value is currently invalid, and aborting the form submission. UAs would typically only do this for the first form control found to be invalid; while the event is dispatched to all successful but invalid controls, it is simpler for the user to deal with one error at a time. If the element causing trouble is not visible (for example a field made invisible using CSS or a field of type `hidden`) then the UA may wish to indicate to the user that there may be an error with the page's script.

Note: This specification currently does not specify what should happen if some events are cancelled and some are not. A future version of this specification may specify this in more detail based on implementation feedback and user experience. Authors are encouraged to either cancel all `invalid` events (if they wish to handle the error UI themselves) or to not cancel any (if they wish to leave the error UI to the UA).

When fired by script calling the `validate()` method (i.e. not during form submission), the event has no default action.

> The following example shows one way to use this event.
>
> ```
> <form action="..." method="post">
>  
>   <label>
>    Byte 1:
>    <input name="byte" type="number" min="0" max="255" required="req
>           oninvalid="failed(event)" />
>   </label>
>   <output name="error"/>
>  
>  <script type="text/javascript"> <![CDATA[
>   function failed(event) {
>     // a control can fail for more than one reason; only report one
>     form.error.value = 'The value is wrong for a reason I did not e
>     if (event.target.validity & event.target.form.ERROR_TYPE_MISMAT
>       form.error.value = 'That is not an integer.';
>     else if (event.target.validity & event.target.form.ERROR_PRECIS
>       form.error.value = 'That is not an integer.';
>     else if (event.target.validity & event.target.form.ERROR_RANGE_
>       form.error.value = 'That integer is less than 0.';
>     else if (event.target.validity & event.target.form.ERROR_RANGE_
>       form.error.value = 'That integer is more than 255.';
>     else if (event.target.validity & event.target.form.ERROR_REQUIR
>       form.error.value = 'You did not enter a value.';
>     event.preventDefault(); /* don't want the UA to do its own repo
>   }
>  ]]> </script>
> </form>
> ```

4.5. Receiving the results of form submission

The **`ReceivedEvent`** interface is used in the form submission process to handle the results of form submission.

```
interface ReceivedEvent : Event {
  readonly attribute Document document;
  void               initReceivedEvent(in DOMString typeArg,
                                       in boolean canBubbleArg,
```

```
                                        in boolean cancelableArg,
                                        in Document documentArg);
 void               initReceivedEventNS(in DOMString namespaceURIArg,
                                        in DOMString typeArg,
                                        in boolean canBubbleArg,
                                        in boolean cancelableArg,
                                        in Document documentArg);
```

The `initReceivedEvent()` and `initReceivedEventNS()` methods have the same behaviours as the `initEvent()` and `initEventNS()` events from [DOM3EVENTS].

The `document` argument contains a reference to the document that was the result of the form submission. If the result cannot be represented as a DOM document, then the attribute is null. The document is mutable.

5. Form submission

Processors conforming to this specification must use a slightly different algorithm than the [HTML4] form submission algorithm (HTML4 section 17.13.3), as described in this section.

When the user agent submits a form, it must perform the following steps.

1. **Step one: Dispatch the `submit` event.**

   If the submission was not initiated using the `submit()` method then the `submit` event is submitted as described in [HTML4]. If it is canceled, then the submission processing stops at this point. If it is not cancelled, then its default action is to perform the rest of the submission procedure.

2. **Step two: Check the validity of the form**

   If the form submission was initiated as a result of a `submit` event's default action, then the form is checked for validity. If this step fails (that is, if any `invalid` events are fired), the submission is aborted.

   Otherwise, if the form submission was initiated via the `submit()` method, then instead of firing `invalid` events, an exception is raised (and submission is aborted) if any of the controls are invalid. Specifically, a `SYNTAX_ERR` exception is raised. [DOM3CORE]

   Script authors who wish to validate the form then perform submission can use script such as:

```
if (form.validate())
   form.submit();
```

...with the controls having event handlers that report the errors.

3. **Step three: Identify all form controls**

   All the controls that apply to the form, whether successful or not, should be taken, in document order. These controls are all those whose `form` DOM attribute points at the form and that are not in the form's `templateElements` DOM attribute (this excludes certain controls as specified in the section describing the repetition model).

4. **Step four: Build a form data set**

   A **form data set** is a sequence of *control-name, index, current-value* triplets constructed from the controls identified in the previous step.

   > ***Note: The index here is unrelated to the repetition index mentioned earlier.***

   It is constructed by iterating over the form controls listed in step three, taking note of the form control names as they are seen. With each control, if it is the first time that control's name has been seen, then the control is assigned an index of 0. Otherwise, if the control name was associated with an earlier control, then the index assigned is exactly one more than the last control with that name. Even unsuccessful controls and controls with no value are so numbered. However, only successful controls are added to the form data set.

   Successful controls have exactly one value, except for `select` controls and file upload controls, which have zero or more values depending on how many items or files they have selected. A successful control with more than one value is added multiple times, one for each value (each time with the same form control name and form control index). A successful control with zero values is omitted from the form data set.

   Image buttons, during this step, are handled as if they were two controls, one with the control's name with `.x` appended, whose value is the x coordinate selected by the user, and the other with the control's name with `.y` appended, whose value is the y coordinate selected by the user. The indices of these two virtual controls are handled separately and could, depending on the values of other controls, end up with different values.

   > For example, the following form:
   >
   > ```
   > <form>
   > ```

```
<p> <label> Name: <input type="text" name="username"/> </labe
<p> Lottery numbers:
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
    <input name="number" type="number" min="1" max="49"/>
</p>
<p>
 <label>
  Games:
  <select name="type" multiple="multiple">
   <option value="Thunderbolt"> Thunderbolt </option>
   <option value="Lightning"> Lightning </option>
  </select>
 </label>
</p>
<p>
 <input type="submit" value="Send">
</p>
</form>
```

...if filled in with the name "Erwin" and the numbers 20, 30 and 40 with the first and last number fields left blank, and all the values in the select list selected, would generate the following form data set:

1. username, 0, "Erwin"
2. number, 0, ""
3. number, 1, "20"
4. number, 2, "30"
5. number, 3, "40"
6. number, 4, ""
7. type, 0, "Thunderbolt"
8. type, 0, "Lightning"

The form data set also includes a list of which repetition blocks are involved in the submission.

For each control in the form data set, the control and the control's ancestors are examined, up to but not including the first node that is a common ancestor of the control and the form, or is the form itself. For each element so examined, if it is a repetition block that is not an orphan repetition block and whose template does have an ID, and that repetition block has not yet been added to the list of repetition blocks, it is added.

5. **Step five: Encode the form data set**

   The form data set is then encoded according to the content type specified by the `method` and `enctype` attribute of the element that caused the form to be submitted. See the semantics of `method` and `enctype` attributes section for details on how the `action` and `enctype` attributes are to be treated. The possible values of `enctype` defined by this specification are:

   `application/x-www-form-urlencoded`

   Described below.

   `multipart/form-data`

   Described in [HTML4], section 17.13.4. Note that this submission method discards the index and repetition block parts of the form data set.

   `application/x-www-form+xml`

   Described below.

   `text/plain`

   Described below.

   Attribute not specified

   Described below.

   Other values may be defined by other specifications.

   During this step, the form data set is examined to ensure all the characters are representable in the submission character encoding.

6. **Step seven: Submit the encoded form data set**

   Finally, the encoded data is sent to the processing agent designated by the `action` attribute of the element that initiated the submission using the protocol method specified by the `method` attribute of that same element. The semantics of `method` and `enctype` attributes section describes this in more detail.

7. **Step eight: Dispatch the `received` event.**

   This step must be skipped if the form has no `onreceived` attribute. If this step is not skipped. then it defeats any attempt at incremental rendering, as the entire return value from the server must be downloaded and parsed before the event is fired (unless the user agent instantiates the document lazily).

   The **`received`** event is fired on the `form` element. This event does not

bubble. The **onreceived** attribute can be used to handle this event.

The event uses the ReceivedEvent interface described below.

If it is canceled, then the submission processing stops at this point. If it is not cancelled, then its default action is to perform the rest of the submission procedure (step nine). If the document attribute of the event was mutated, the mutated version is what is used in the next step.

8. **Step nine: Handle the returned data**

   If the response is an HTTP 204 No Content response (or equivalent for other protocols), then the document is left in place, and new metadata (if any) is applied. as per the HTTP specification [RFC2616].

   Otherwise, how the UA responds to a response depends on the replace attribute of the element that initiated the submission.

   For replace="document" (the default), the response body replaces the document from which the submission initiated (or, if there is a target attribute, the document in the appropriate frame).

   > For example if the action denotes an HTTP resource, method is "POST", the replace attribute is document and the remote server replies with a 200 OK response, then the returned document should be displayed to the user as if the user had navigated to that document by following a link to it.

   For replace="values", the algorithm described in the section on seeding a form with initial values must be run with the given response body used instead of the document mentioned in the data attribute. (Any target attribute is ignored.)

5.1. Successful form controls

All form controls are successful except:

- Controls with no associated form.
- Controls that are in their form's templateElements list (those inside repetition templates).
- Controls with no name.
- Disabled controls.
- Checkboxes that are not checked.

- Radio buttons that are not checked.
- Submit buttons (including image buttons) that did not initiate the current submission process.
- Buttons of type `button`, `reset`, `add`, `remove`, `move-up`, or `move-down`.
- Fieldsets.
- Output controls.
- File upload controls with no value.
- File upload controls with a value that is not an existent file.

Controls do not have to have a value to be successful.

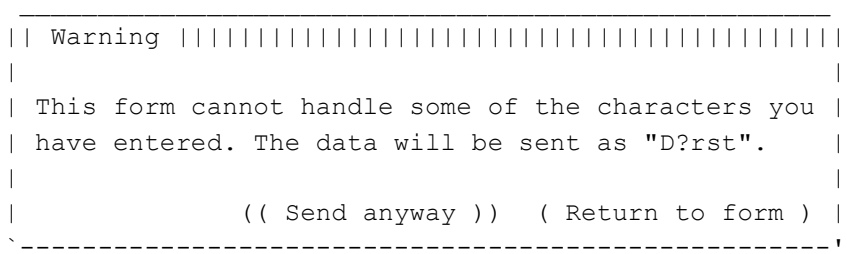
Unsuccessful controls are not checked for validity during submission (although their flags are still set appropriately).

5.2. Handling characters outside the submission character set

The different form data set encoding types each define how to find the character encoding to use to submit the data.

Sometimes, the form submission character set used is not able to represent all the character present in the form submission.

If the form data set contains characters that are outside the submission character set, the user agent should inform the user that his submission will be changed, for example using a dialog in the form:



If the submission is not canceled, the user agent MUST replace each character that is not in the submission character set with one or more replacement characters.

For each such missing character, UAs must either transliterate the character to a UA-defined human-recognizable representation (for example transliterating U+263A to the three-character string ":-)" in US-ASCII, or U+2126 to the byte 0xD9 in ISO-8859-7), or, for characters where a dedicated transliteration is not

known to the UA, replace the character with either U+FFFD, "?", or some other single character representing the same semantic as U+FFFD.

Note that a string containing the codepoint's value itself (for example the six-character string "U+263A" or the seven-character string "☺") is not considered to be human readable and must not be used as a transliteration. (This is to discourage servers from attempting to mechanically convert such codepoints back into Unicode characters, as there is no way to distinguish such characters from identical literal strings entered by the user.)

5.3. `application/x-www-form-urlencoded`

This section defines the expected behaviour for step 5, "Step five: Encode the form data set", of the submission algorithm described above, for the form content type `application/x-www-form-urlencoded`. The rest of the form submission process progresses as described above.

This is the default content type. Forms submitted with this content type must be encoded as follows:

1. The submission character encoding is selected from the form's `accept-charset` attribute. UAs must use the encoding that most completely covers the characters found in the form data set of the encodings specified. If the attribute is not specified, then the client should use either the page's character encoding, or, if that cannot encode all the characters in the form data set, UTF-8. Character encodings that are not mostly supersets of US-ASCII must not be used (this includes UTF-16 and EBCDIC) even if specified in `accept-charset` attribute.

   How a UA establishes the page's character encoding is determined by the language specification. It could be explicitly specified by the page, overriden by the user, or auto-detected by the UA. For example, HTML4 section 5.2.2 [HTML4].

2. If the form contains an input control of type `hidden` with the name `_charset_`, it is forced to appear in the form data set, with the value equal to the name of the submission character encoding used.

3. The values of file upload controls are the names of the files selected by the user, *not* their contents.

4. Control names and values are escaped. Space characters are replaced by `+', and then non-alphanumeric characters are encoded in the submission character encoding and each resulting byte is replaced by `%HH', a percent sign and two uppercase hexadecimal digits representing the value of the byte.

5. The control names/values are listed in the order they appear in the form data set. The name is separated from the value by `=' and name/value pairs are separated from each other by `&'.

Note that the index and repetition block parts of the form data set are not used.

5.4. `application/x-www-form+xml`: XML submission

This section defines the expected behaviour for step 5, "Step five: Encode the form data set", of the submission algorithm described above, for the form content type `application/x-www-form+xml`. The rest of the form submission process progresses as described above.

The message entity is an XML 1.1 document, encoded as UTF-8, which has a root element named "submission", with no prefix, defining a default namespace `data:,formData`. UAs must include an XML declaration.

Note that the form's `accept-charset` attribute is ignored for this encoding type.

First, for each repetition block in the form data set, an element `repeat` is inserted, with an attribute `template` equal to the ID of the template, and an attribute `index` equal to the index of the repetition block. The element is empty.

Servers are generally expected to ignore `repeat` elements; they are primarily included so that form data can be round-tripped using the `data` attribute on the form element.

Then, for each successful control that is not a file upload control, in the order that the controls are to be found in the original document, an element `field` is inserted, with an attribute `name` having the name of the form control, an attribute `index` having the index described above in the definition of the form data set, and with the element content being the current value of the form control. Form controls with multiple values result in multiple `field` elements being inserted into the output, one for each value, all with the same index.

File controls are submitted using a `file` element instead of a `field` element. The `file` element has four attributes, `name`, `index`, `filename`, and `type`. The `name` attribute contains the name of the file control. The `index` attribute contains the index in the control's entry in the form data set. The `filename` attribute is optional and may contain the name of the file. The `type` attribute is also optional and must contain the MIME type of the file, or be omitted if the client is unaware of the correct type. The type may contain MIME parameters if appropriate. The contents of the file are base64 encoded and then included literally as content directly inside the `file` element. As base64 data is whitespace-clean, UAs may introduce whitespace into the `file` element to ensure the submitted data has reasonable line lengths. This is, however, completely optional. (It is primarily

intended to make it possible to write readable examples of submission output.)

UAs may use either CDATA blocks, entities, or both in escaping the contents of attributes and elements, as appropriate. The resulting XML must be a well-formed XML instance. The only mention of namespaces in the submission document must be the declaration of the default namespace on the root element.

Whitespace may be inserted around elements that are children of the `submission` element in order to make the submitted data easier to scan by eye. However, this is optional. Processors should not be affected by such whitespace, or whitespace inside `file` elements, when reading the submitted data back from the XML instance. (Whitespace inside `field` elements is significant, however.)

> The following example illustrates `application/x-www-form+xml` encoding. Suppose we have the following form:
>
> ```
> <form action="http://example.com/cgi/handle"
>       enctype="application/x-www-form+xml"
>       method="post">
>  
>   <label> What is your name? <input type="text" name="submit-name"/
>   <label> What files are you sending? <input type="file" name="file
>   <label> When were they written? <input type="date" name="stamp"/>
>   <input type="submit" value="Send">
>  
> </form>
> ```
>
> If the user enters "Larry" in the text input, selects the text file "file1.txt", and picks an arbitrary date, the user agent might send back the following data:
>
> ```
> Content-Type: application/x-www-form+xml
>
> <submission xmlns="data:,formData">
>  <field name="submit-name" index="0">Larry</field>
>  <file name="files" index="0" filename="file1.txt" type="text/plain
>   Y29udGVudHMgb2YgZmlsZTEudHh0
>  </file>
>  <field name="stamp" index="0">1979-04-13</field>
> </submission>
> ```
>
> If the user selected a second (image) file "file2.png", and changes the date, the user agent might construct the entity as follows:
>
> ```
> Content-Type: application/x-www-form+xml
>
> <submission xmlns="data:,formData">
>  <field name="submit-name" index="0">Larry</field>
>  <file name="files" index="0" filename="file1.txt" type="text/plain
>   Y29udGVudHMgb2YgZmlsZTEudHh0
> ```

```
 </file>
 <file name="files" index="0" filename="file2.png" type="image/png"
  iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAMAAAAoyzS7AAAABGdBTUEAAK
  /INwWK6QAAABl0RVh0U29mdHdhcmUAQWRvYmUgSW1hZ2VSZWFkeXHJZTwA
  AAAGUExURQD/AAAAG8DfkMAAAAMSURBVHjaYmAACDAAAAIAAU9tWeEAAA
  AASUVORK5CYII=
 </file>
 <field name="stamp" index="0">1979-12-27</field>
</submission>
```

Note how the content of the plain text attached file is base64-encoded, despite being a plain text file. This preserves the integrity of the file in cases where the MIME type is incorrect. It also means that files with malformed content, for example a file encoded as UTF-8 with stray continuation bytes, will be transmitted faithfully instead of being re-encoded by the UA.

This example illustrates this encoding for the case with two form controls with the same name. Suppose we have the following form:

```
<form enctype="application/x-www-form+xml" method="post">
 <p>
  Enter your new password twice:
  <input type="password" name="password"/>
  <input type="password" name="password"/>
  <input type="submit" value="Send">
 </p>
</form>
```

If the user enters "perfect" and "prefect", the user agent might send back the following data:

```
Content-Type: application/x-www-form+xml

<submission xmlns="data:,formData">
 <field name="password" index="0">perfect</field>
 <field name="password" index="1">prefect</field>
</submission>
```

Recall the example for repetition blocks. If it was immediately submitted, the output would be an XML file equivalent to:

```
Content-Type: application/x-www-form+xml

<submission xmlns="data:,formData">
 <repeat template="row" index="0"/>
 <field name="name_0" index="0">John Smith</field>
 <field name="count_0" index="0">2</field>
 <field name="name_1" index="0"></field>
 <field name="count_1" index="0">1</field>
</submission>
```

5.5. `text/plain`

This section defines the expected behaviour for step 5, "Step five: Encode the form data set", of the submission algorithm described above, for the form content type `text/plain`. The rest of the form submission process progresses as described above.

This content type is more human readable than the others but is not unambiguously parseable. Forms submitted with this content type must be encoded as follows:

1. The submission character encoding is selected from the form's `accept-charset` attribute. UAs must use the encoding that most completely covers the characters found in the form data set of the encodings specified. If the attribute is not specified, then the client should use either the page's character encoding, or, if that cannot encode all the characters in the form data set, UTF-8.

   How a UA establishes the page's character encoding is determined by the language specification. It could be explicitly specified by the page, overriden by the user, or auto-detected by the UA. For example, HTML4 section 5.2.2 [HTML4].

2. If the form contains an input control of type `hidden` with the name `_charset_`, it is forced to appear in the form data set, with the value equal to the name of the submission character encoding used.

3. The values of file upload controls are the names of the files selected by the user, *not* their contents.

4. The control names/values are listed in the order they appear in the form data set. The name is separated from the value by `=' and name/value pairs are separated from each other by a newline character.

Note that the index and repetition block parts of the form data set are not used.

> ***Note: This algorithm does not directly parallel the algorithm for `application/x-www-form-urlencoded`. This is mostly due to backwards compatibility concerns.***

5.6. Semantics of `method` and `enctype` attributes

The exact semantics of the `method` and `enctype` attributes depend on the protocol specified by the `action` attribute, in the manner described in this section.

The attributes considered are those of the element that initiated the submission — if the user started the submission then the attributes come from the submit button or image that the user activated; if script started the submission then the attributes of the form are used. If an attribute is missing from a submit button, then the equivalent attribute on the form is used instead.

> In the following example:
>
> ```
> <form action="test.php" method="post">
>  <input type="submit">
>  <input type="submit" method="get">
> </form>
> ```
>
> The first submit button would submit to the `test.php` script using the HTTP POST method, and the second would submit to the same script but using the HTTP GET method.

The HTTP specification defines various methods that can be used with HTTP URIs. Four of these are allowed as values of the `method` attribute: `get`, `post`, `put`, and `delete`. In this specification, these method names are applied to other protocols as well. This section defines how they should be interpreted.

If the specified `method` is not one of `get`, `post`, `put`, or `delete` then it is treated as `get` in the tables below.

If the `enctype` attribute is not specified (or is set to the empty string), and the form consists of exactly one file upload control with exactly one file selected, then in the tables below, the "File upload" rows should be used. If the form contains more than just one file upload control with exactly one file selected, or if the attribute *is* specified but has on unrecognised value, the `enctype` attribute is treated as if it was `application/x-www-form-urlencoded`.

User agents may implement whichever URI schemes are required for their particular application. This specification does not specify a required core set of protocols that must be implemented.

What user agents should do when the designated resource is fetched depends on the value of the `replace` attribute. This is described in step nine of the algorithm.

5.6.1. For `http:` actions

HTTP is described by [RFC2616].

	get	post	put	delete
application/x-www-form-urlencoded	Use the encoded data set as the query value for a URI formed from	Use the encoded data set as the entity body, with the		Ignore the form data set and access

	the `action` URI and fetch it via HTTP GET.	`Content-Type` set appropriately, and submit it using the specified method.	`action` with the specified method.
multipart/form-data	Handle as if `enctype` was `application/x-www-form-urlencoded`.		
application/x-www-form+xml			
text/plain			
File upload		Use the file content as the entity body, with the `Content-Type` set to its MIME type, and submit it using the specified method.	

5.6.2. For `ftp:` actions

The `ftp:` URI scheme is described by [RFC1738] and FTP itself is described by [RFC959].

Using the FTP protocol for form submission is of dubious value and is discouraged.

	get	post	put	delete
application/x-www-form-urlencoded	Ignore the form data set and retrieve the file specified by `action` (`RETR`).	Handle as if `method` was `put`.	Use the encoded data set as the content of a file and upload it to the location specified by `action` (`STOR`). The response body has no content (equivalent to an HTTP 204 No Content response.)	Ignore the form data set and delete the file specified by `action` (`DELE`). The response body has no content (equivalent to an HTTP 204 No Content response.)
multipart/form-data				
application/x-www-form+xml				
text/plain				
File upload			Upload the selected file to the location specified by the `action` URI (`STOR`). The response body has no content (equivalent to an HTTP 204 No Content response.)	

> Using these semantics, a poor man's FTP upload form could be written like so:

```
<form method="put" xmlbase="ftp://ftp.example.com/incoming/">
 <p>
  <legend>
   Path:
   <input type="text" pattern="[^./][^/]*"
          onchange="if (validity == 0) form.action = encodeURICo
  </legend>
  <input type="file" name="file"/>
  <input type="submit" value="Upload file"/>
 </p>
</form>
```

5.6.3. For `data:` actions

The `data:` URI scheme is described by [RFC2387].

	get	post	put	delete
application/x-www-form-urlencoded	Ignore the form data set and access the `action` URI.	URI escape the encoded form data set and substitute it for the first occurance of the string '%%' in the `action` (if any), then access the resulting URI.	Ignore `action`, and form a new data: URI from the entity body, using the appropriate MIME type.	Handle as if `method` was `post`.
multipart/form-data	Handle as if `enctype` was `application/x-www-form-urlencoded`.			
application/x-www-form+xml				
text/plain				
File upload			Ignore `action`, and form a new data: URI from the selected file's contents, using the file's MIME type.	

Note that '%%' is invalid in a URI, so authors should exercise caution when using the `post` method with `data:` URIs..

5.6.4. For `file:` actions

The `file:` URI scheme is described by [RFC1738].

For security reasons, untrusted content should never be allowed to submit or fetch files specified by `file` URIs.

The semantics described in this subsection are recommended, but UAs may implement alternative semantics as consistent behaviour for submission to `file:` URIs is not required for interoperability on the World Wide Web.

<table>
<tr><th></th><th>get</th><th>post</th><th>put</th><th>delete</th></tr>
<tr><td>application/x-www-form-urlencoded</td><td rowspan="4">Ignore the form data set and retrieve the file specified by <code>action</code>.</td><td rowspan="3">If the specified file is executable, launch the specified file in an environment that complies to the CGI Specification [CGI], using the encoded data set as the input, and the resulting standard output as the response body.</td><td rowspan="3">Use the encoded data set as the content of a file and store it in the location specified by <code>action</code>. The response body has no content (equivalent to an HTTP 204 No Content response.)</td><td rowspan="4">Ignore the form data set and delete the file specified by <code>action</code>. The response body has no content (equivalent to an HTTP 204 No Content response.)</td></tr>
<tr><td>multipart/form-data</td></tr>
<tr><td>application/x-www-form+xml</td></tr>
<tr><td>text/plain</td></tr>
<tr><td>File upload</td><td>Handle as for other types except the encoded form data set is the contents of the specified file.</td><td>Store the selected file at the location specified by the <code>action</code> URI.</td></tr>
</table>

5.6.5. For `mailto:` actions

The `mailto:` URI scheme is described by [RFC2368].

UAs should not send e-mails without the explicit consent of the user.

All submissions made using `mailto:` result in the equivalent of an HTTP 204 No Content response. Thus the `replace` attribute is effectively ignored when `enctype` is a `mailto` URI.

<table>
<tr><th></th><th>get</th><th>post</th><th>put</th><th>delete</th></tr>
<tr><td>application/x-www-form-urlencoded</td><td>Use the encoded data set as the <code>headers</code> part (see [RFC2368]) of a <code>mailto:</code> URI formed from the <code>action</code> URI and process that URI.</td><td rowspan="4">Use the encoded data set as the default message body, with the <code>Content-Type</code> set appropriately, for a message based on the specified <code>action</code> URI.</td><td colspan="2" rowspan="4">Handle as if <code>method</code> was <code>post</code>.</td></tr>
<tr><td>multipart/form-data</td><td rowspan="3">Handle as if <code>enctype</code> was <code>application/x-www-form-urlencoded</code>.</td></tr>
<tr><td>application/x-www-form+xml</td></tr>
<tr><td>text/plain</td></tr>
</table>

File upload		Attach the selected file to a message based on the specified `action` URI.	

5.6.6. For `smsto:` and `sms:` actions

The `smsto:` and `sms:` URI schemes are not yet specified.

UAs should not send SMSes without the explicit consent of the user.

All submissions made using the `smsto:` and `sms:` URI schemes result in the equivalent of an HTTP 204 No Content response. Thus the `replace` attribute is effectively ignored when `enctype` is an SMS URI.

	get	post	put	delete
application/x-www-form-urlencoded	*Behaviour is undefined, pending the release of an `smsto:` or `sms:` specification.*	Use the encoded data set as the default message body for a message based on the specified `action` URI.	Handle as if `method` was `post`.	
multipart/form-data	Handle as if `enctype` was `application/x-www-form-urlencoded`.	Handle as if `enctype` was `application/x-www-form-urlencoded`.		
application/x-www-form+xml				
text/plain		Use the encoded data set as the default message body for a message based on the specified `action` URI.		
File upload		Handle as if `enctype` was `application/x-www-form-urlencoded`.		

5.6.7. For `javascript:` actions

The `javascript:` URI scheme is described by [CSJSR]. ECMAScript is defined in [ECMA262].

If the response body of a submission to a `javascript:` action is the ECMAScript `void` type, then it is treated as if it was an HTTP 204 Not Content response.

	get	post	put	delete
application/x-www-form-urlencoded	Ignore the form data set and access the `action`	Encode the form data set by putting each name/value pair into a newly created object using the names as attributes of	Handle as if `method` was `post`.	

multipart/form-data	URI in the current context. The response body is the return value of the script.	that object and the values as the values of those attributes. Execute the URI in the context of the document after having added the aforementioned object to the start of the scope chain. Duplicate names should cause the property to become an array, with each value represented in the array. The response body is the return value of the script.	
application/x-www-form+xml			
text/plain			
File upload			

6. Fetching data from external resources

There are two scenarios where authors may wish data to be fetched from an external file to fill forms. In the first, a `select`'s options are replaced by options from an external file. In the second, a form's values are prefilled from an external data source.

In both cases, the prefilling may either be full, in which case the previous contents are removed first, or incremental, in which case the fetched data is in addition to the data already in the form.

Implementations may limit which hosts, ports, and schemes can be accessed using these methods. For example, HTTP-based content should not be able to pre-seed a form based on content from the local file system. Similarly, cross-domain scripting restrictions are fully expected to apply.

6.1. Filling `select` elements

If a `select` element being parsed has a `data` attribute, then as soon as the `select` element and all its children have been parsed and added to the document, it should be prefilled.

If a `select` element has a `data` attribute, it must be a URI that points to a well-formed XML file whose root element is a `select` element in the `http://www.w3.org/1999/xhtml` namespace. The MIME type must be an XML MIME type [RFC3023], preferably `application/xml`. It should not be `application/xhtml+xml` since the root element is not `html`.

UAs must process this file if it has an XML MIME type [RFC3023], if it is a well-formed XML file, and if the root element is the right root element in the right namespace. If any of these conditions are not met, UAs must act as if the attribute was not specified, although they may report the error to the user. UAs

are expected to correctly handle namespaces, so the file may use prefixes, etc.

If the UA processes the file, it must use the following algorithm to fill the form.

1. Unless the root element has a `type` attribute with the exact literal string `incremental`, the children of the `select` element in the original document must all be removed from the document.

2. The entire contents of the `select` element in the referenced document are imported into the original document and inserted as children of the `select` element. (Even if importing into a `text/html` document, the newly imported nodes will still be namespaced.)

3. All nodes outside the `select` are ignored, as are attributes on the `select`.

If a `select` element has its `data` attribute manipulated via the DOM, then that should immediately begin the prefilling process too. However, any currently executing script must be guaranteed to run to completion before the changes required by the process take effect. If the process is started while an outstanding prefilling request is still being attended to, the requests must all be serviced in the order they were started.

> The following script has only one possible valid outcome:
>
> ```
> var select = document.createElementNS('http://www.w3.org/1999/xhtml
> select.data = 'data:application/xml,<select xmlns="http://www.w3.or
> select.data = select.data;
> // at this point, select.length == 0 is guaranteed
> var option = document.createElementNS('http://www.w3.org/1999/xhtml
> option.appendChild(document.createTextNode('a'));
> select.appendChild(option);
> // at this point, select.length == 1 is guaranteed
> document.documentElement.appendChild(select);
> ```
>
> ...namely, the insertion at the end of the document of a select widget which, in due course, will have three options, namely 'a', 'b' and 'b'. Note that if the script was modified so tha the URI in the second line did not say `type="incremental"`, then the resulting select widget would only have one option, the last 'b'.

6.2. Seeding a form with initial values

Before `load` events are fired, but after the entire document has been parsed and after `select` elements have been filled from external data sources (if necessary), forms with `data` attributes are prefilled.

Note: In particuar, UAs should not specifically wait for images and

stylesheets to be loaded before preseeding forms.

If a `form` has a `data` attribute, it must be a URI that points to a well-formed XML file whose root element is a `formdata` element in the `data:,formData` namespace. The MIME type must be an XML MIME type [RFC3023], preferably `application/xml`.

UAs must process this file if it has an XML MIME type [RFC3023], if it is a well-formed XML file, and if the root element is the right root element in the right namespace. If any of these conditions are not met, UAs must act as if the attribute was not specified, although they may report the error to the user. UAs are expected to correctly handle namespaces, so the file may use prefixes, etc.

If the UA processes the file, it must use the following algorithm to fill the form.

1. Unless the root element has a `type` attribute with the exact literal string `incremental`, the form must be reset to its initial values as specified in the markup.

2. Child text nodes, CDATA blocks, comments, and PIs of the root element of the specified file must be ignored.

3. `repeat` elements in the `data:,formData` namespace that are children of the root element, have a non-empty `template` attribute and an `index` attribute that contains only one or more digits in the range 0-9 with an optional leading minus sign, have no other non-namespaced attributes, and have no content, must be processed as follows:

   If the `template` attribute specifies an element that is not a repetition template, then the element is ignored.

   If the `template` attribute specifies a repetition template and that template already has a repetition block with the index specified by the `index` attribute, then the element is ignored.

   Otherwise, the specified template's `addRepetitionBlockByIndex()` method is called, with a null first argument and the index specified by the `repeat` element's `index` attribute as the second.

4. `field` elements in the `data:,formData` namespace that are children of the root element, have a non-empty `name` attribute and an `index` attribute that contains only one or more digits in the range 0-9, have no other non-namespaced attributes, and have either nothing or only text and CDATA nodes as children, must be used to initialize fields, as follows.

   First, the form control that the field references must be identified. This is

done by walking the list of form controls associated with the form until one is found that has a name exactly equal to the name given in the `field` element's `name` attribute, skipping as many such matches as is specified in the `index` attribute.

If the identified form control is a file upload control, a push button control, or an image control, then the `field` element is now skipped.

Next, if the identified form control is not a multiple-valued control (a multiple-valued control is one that can generate more than one value on submission, such as a `<select multiple="multiple">`), or if it is a multiple-valued control but it is the first time the control has been identified by a `field` element in this data file that was not ignored, then it is set to the given value (the contents of the `field` element), removing any previous values (even if these values were the result of processing previous `field` elements in the same data file). Otherwise, this is a subsequent value for a multiple-valued control, and the given value (the contents of the `field` element) should be *added* to the list of values that the element has selected.

If the element cannot be given the value specified, the `field` element is ignored and the control's value is left unchanged. For example, if a checkbox has its value attribute set to `green` and the `field` element specifies that its value should be set to `blue`, it won't be changed from its current value. (The only values that would have an effect in this example are "", which would uncheck the checkbox, and "green", which would check the checkbox.) Another example would be a `datetime` control where the specified value is outside the range allowed by the `min` and `max` attributes. The format must match the allowed formats for that type for the value to be set.

If the element is a multiple-valued control and the control already has the given value selected, but it can be given the value again, then that occurs. For example, in the following case:

```
<select name="select" multiple="multiple">
 <option>test</option>
 <option>test</option>
 <option>test</option>
</select>
```

...if the data file contained two instances of:

```
<field name="select" index="0">test</select>
```

...then the first two `option` elements would end up selected, and the last would not. This would be the case irrespective of which `option` elements

had their `selected` attribute set in the markup.

Note: The `option` elements are never directly matched by `field` elements; it is the `select` element in this case that is matched (twice). This is why the two `field` elements select subsequent values in the control.

If the element is a multiple-valued control and the control already has the given value selected and it *cannot* be given the value again, then the field is ignored.

5. All other elements in the file must be ignored.
6. A `formchange` event is then fired on all the form controls of the form.

Note: Note that file upload controls cannot be repopulated. However, `output` control can be populated. This can be used, for example, for localizing a form by including the structure in one file and the strings in another. (The semantics of this practice are somewhat dubious, however. It is only mentioned because XForms advocates claim this as a feature.)

Setting the `data` attribute dynamically does not cause the UA to refill the form. The semantics of the `data` attribute are only relevant during initial document load, with the form filling kicked off just as the document has finished being parsed. Thus, forms with `data` attributes that are added to the document by script before the UA has finished parsing the document must be prefilled. The DOM can be used to refill a form after the document has finishing loading.

7. Extensions to the HTML Level 2 DOM interfaces

Unless otherwise specified, these interfaces have the same semantics as defined in [DOM2HTML].

```
interface HTMLFormElement : HTMLElement {
  readonly attribute HTMLCollection  elements;
  readonly attribute long            length;
           attribute DOMString       name;
           attribute DOMString       acceptCharset;
           attribute DOMString       action;
           attribute DOMString       enctype;
           attribute DOMString       method;
           attribute DOMString       target;
```

```
  void               submit();
  void               reset();

  // new in this specification:
  const    unsigned short            ERROR_TYPE_MISMATCH      = 1;
  const    unsigned short            ERROR_RANGE_UNDERFLOW    = 2;
  const    unsigned short            ERROR_RANGE_OVERFLOW     = 4;
  const    unsigned short            ERROR_PRECISION_EXCEEDED = 8;
  const    unsigned short            ERROR_TOO_LONG           = 16;
  const    unsigned short            ERROR_PATTERN_MISMATCH   = 32;
  const    unsigned short            ERROR_REQUIRED           = 64;
  const    unsigned short            ERROR_USER_DEFINED       = 32768;

           attribute DOMString       accept;
           attribute DOMString       replace;
  readonly attribute HTMLCollection  templateElements;
  bool               validate();
  bool               willConsiderForSubmission(in Element element);
  void               resetFromData(in Document data);

  void               dispatchFormInput();
  void               dispatchFormChange();
};

interface HTMLSelectElement : HTMLElement {
  readonly attribute DOMString       type;
           attribute long            selectedIndex;
           attribute DOMString       value;
           attribute unsigned long   length;
                                        // raises(DOMException) on sett

  readonly attribute HTMLFormElement form;
  readonly attribute HTMLOptionsCollection options;
           attribute boolean         disabled;
           attribute boolean         multiple;
           attribute DOMString       name;
           attribute long            size;
           attribute long            tabIndex;
  void               add(in HTMLElement element,
                         in HTMLElement before)
                                        raises(DOMException);
  void               remove(in long index);
  void               blur();
  void               focus();

  // new in this specification:
           attribute DOMString       pattern;
           attribute boolean         required;
           attribute boolean         autocomplete;
           attribute DOMString       inputmode;
           attribute long            maxLength;
  readonly attribute HTMLCollection  selectedOptions;
  readonly attribute HTMLCollection  labels;
```

```
  readonly attribute boolean         successful;
  readonly attribute long            validity;
  bool               validate();
  void               setValidity(in boolean valid);
  void               changed();
  void               formchanged();
};

interface HTMLOptGroupElement : HTMLElement {
           attribute boolean         disabled;
           attribute DOMString       label;
};

interface HTMLOptionElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute boolean         defaultSelected;
  readonly attribute DOMString       text;
  readonly attribute long            index;
           attribute boolean         disabled;
           attribute DOMString       label;
           attribute boolean         selected;
           attribute DOMString       value;
};

interface HTMLInputElement : HTMLElement {
           attribute DOMString       defaultValue;
           attribute boolean         defaultChecked;
  readonly attribute HTMLFormElement form;
           attribute DOMString       accept;
           attribute DOMString       accessKey;
           attribute DOMString       align;
           attribute DOMString       alt;
           attribute boolean         checked;
           attribute boolean         disabled;
           attribute long            maxLength;
           attribute DOMString       name;
           attribute boolean         readOnly;
           attribute unsigned long   size;
           attribute DOMString       src;
           attribute long            tabIndex;
           attribute DOMString       type;
           attribute DOMString       useMap;
           attribute DOMString       value;
  void               blur();
  void               focus();
  void               select();
  void               click();

  // new in this specification:
           attribute DOMString       min;
           attribute DOMString       max;
           attribute DOMString       pattern;
           attribute boolean         required;
           attribute boolean         autocomplete;
```

```
           attribute DOMString       inputmode;
  readonly attribute RepetitionElement template;
  readonly attribute HTMLCollection  labels;

           attribute DOMString       action;
           attribute DOMString       enctype;
           attribute DOMString       method;
           attribute DOMString       replace;
  readonly attribute boolean         successful;
  readonly attribute long            validity;
  bool               validate();
  void               setValidity(in boolean valid);
  void               changed();
  void               formchanged();
};

interface HTMLTextAreaElement : HTMLElement {
           attribute DOMString       defaultValue;
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute long            cols;
           attribute boolean         disabled;
           attribute DOMString       name;
           attribute boolean         readOnly;
           attribute long            rows;
           attribute long            tabIndex;
  readonly attribute DOMString       type;
           attribute DOMString       value;
  void               blur();
  void               focus();
  void               select();

  // new in this specification:
           attribute DOMString       wrap;
           attribute DOMString       pattern;
           attribute boolean         required;
           attribute boolean         autocomplete;
           attribute DOMString       inputmode;
           attribute long            maxLength;
  readonly attribute HTMLCollection  labels;

  readonly attribute boolean         successful;
  readonly attribute long            validity;
  bool               validate();
  void               setValidity(in boolean valid);
  void               changed();
  void               formchanged();
};

interface HTMLButtonElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute boolean         disabled;
           attribute DOMString       name;
```

```
           attribute long            tabIndex;
           attribute DOMString       value;

  // modified in this specification
           attribute DOMString       type;

  // new in this specification:
           attribute DOMString       action;
           attribute DOMString       enctype;
           attribute DOMString       method;
           attribute DOMString       replace;
  readonly attribute HTMLCollection  labels;
  readonly attribute RepetitionElement template;
};

interface HTMLLabelElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute DOMString       htmlFor;

  // new in this specification:
  readonly attribute Element         control;
};

interface HTMLFieldSetElement : HTMLElement {
  readonly attribute HTMLFormElement form;

  // new in this specification
           attribute boolean         disabled;
};

interface HTMLLegendElement : HTMLElement {
  readonly attribute HTMLFormElement form;
           attribute DOMString       accessKey;
           attribute DOMString       align;
};

// new in this specification
interface HTMLOutputElement : HTMLElement {
           attribute DOMString       defaultValue;
  readonly attribute HTMLFormElement form;
           attribute DOMString       name;
           attribute DOMString       value;
};

// new in this specification
interface RepetitionElement {
  const              unsigned short  REPETITION_NONE = 0;
  const              unsigned short  REPETITION_TEMPLATE = 1;
  const              unsigned short  REPETITION_BLOCK = 2;

           attribute unsigned short  repetitionType;
           attribute long            repetitionIndex;
  readonly attribute Element         repetitionTemplate;
```

```
  readonly attribute HTMLCollection  repetitionBlocks;
  void               addRepetitionBlock(in Node refNode);
  void               addRepetitionBlockByIndex(in Node refNode, in long
  void               moveRepetitionBlock(in long distance);
  void               removeRepetitionBlock();
}

// new in this specification
interface FormImplementation {
  Document           load(in DOMString action,
                          in EventListener load, in EventListener error
  Document           loadWithBody(in DOMString action, in DOMString met
                                  in DOMString enctype, in DOMString cc
                                  in EventListener load, in EventLister
  Document           loadWithDocument(in DOMString action, in DOMStrinc
                                      in Document document,
                                      in EventListener load,
                                      in EventListener error);
}
```

7.1. Additions specific to the `HTMLFormElement` interface

The new `accept` attribute reflects the `form` element's `accept` attribute and its addition here merely addresses an oversight in DOM2.

The `elements` array is defined to not include image controls (`input` elements of type `image`). This is for backwards compatibility with DOM Level 0. This excludes image buttons from several features of this specification, such as `onformchange` processing and validation.

The `templateElements` attribute contains the list of form controls associated with this form that form part of repetition templates that the form itself is not also a part of. It is defined in more detail in the section on the repetition model. (Image controls *are* part of this array, when appropriate.)

The **`form.validate()`** method invokes the `validate()` method of all the elements in the form's `elements` list whose interfaces have a `validate()` method defined and a `successful` attribute defined and whose `successful` attribute has the true value, and returns the logical-and of all the return values (i.e. it returns false if any of the form controls are successful but invalid).

The **`willConsiderForSubmission()`** must return true if the element passed as an argument is in the form's `elements` array, and is of a type that can be involved in submission. (For example, a text field or radio button control, but not a button of type `button` or a fieldset.) It must also return true for image buttons associated with the form that are not in the `templateElements` list. For all other element it must return false.

> The following event handler checks to see if the event target is a control that will be considered for submission.
>
> ```
> function focussed(event) {
>   if (event.target.form instanceof HTMLFormElement &&
>       event.target.form.willConsiderForSubmission(event.target)) {
>     // event.target is a form control that could be submitted
>   }
> }
> ```

The `reset()` method resets the form, then fires a a `formchange` event on all the form controls of the form.

The **`resetFromData()`** method takes one argument, a document to use for resetting the form. If this argument is null, the method does nothing. Otherwise, the algorithm described in the section on seeding a form with initial values must be run with the given document instead of the document mentioned in the `data` attribute.

The **`dispatchFormChange()`** and **`dispatchFormInput()`** methods cause `formchange` and `forminput` events (respectively) to be fired to all the controls in the `elements` array, much like happens for the default action of `change` and `input` events.

In the ECMAScript binding, objects that implement the `HTMLFormElement` interface reflect their `elements` according to the following semantics:

If a name is used by more than one control, the form object has a property of that name that references a `NodeList` interface that lists the controls of that name.

If a name is used by exactly one control, the form object has a property of that name that references that control.

7.2. Additions specific to the `HTMLSelectElement` interface

The **`selectedOptions`** attribute provides a readonly list of the descendant `HTMLOptionElement` nodes that currently have their `selected` attribute set to a true value (a subset of the controls listed in the `options` attribute). The list is returned live, so changing the options selected (by the user or by script) will change the list. The order of the list should be consistent with the order of the `options` list.

7.2.1. The `HTMLCollection` interface

This specification does not change the `HTMLCollection` interface's definition from its DOM2 HTML definition, but does slightly amend its ECMAScript binding.

For the ECMAScript binding, when `namedItem()` would match more than one node, instead of returning an arbitrary node from the collection, it must return a `NodeList` giving all the nodes that would match, in document order.

> ***Note: This intentionally matches what existing implementations have done. User agents have found that supporting the DOM2 definition strictly is not possible without sacrificing compatibility with existing content.***

7.3. The `HTMLOutputElement` interface

This interface is added for the new `output` element. Its attributes work analogously to those on other controls. The semantics of the `value` and `defaultValue` DOM attributes are described in the section describing the `output` element.

7.4. Validation APIs

The **`successful`** attribute returns whether or not the form control is successful. Only successful controls are included when a form is submitted.

The **`validity`** attribute returns whether the form control is currently valid. Its value is a bit field giving the errors that currently apply ot the control, and must be equal to the sum of the relevant `ERROR_*` constants defined on the `HTMLFormElement` interface. These have the following meanings:

ERROR_TYPE_MISMATCH

The data entered does not match the type of the control. For example, if the UA allows uninterpreted arbitrary text entry for `expdate` fields, and the user has entered `SEP02`, then this error code would be used. This code is also used when the selected file in a file upload control does not have an appropriate MIME type. If the control is empty, this flag will not be set.

ERROR_RANGE_UNDERFLOW

The numeric, date, or time value of a field with a `min` attribute is lower than the minimum, or a file upload control has fewer files selected than the minimum. If the control is empty, this flag will not be set.

ERROR_RANGE_OVERFLOW

The numeric, date, or time value of a field with a `max` attribute is higher than the maximum, or a file upload control has more files selected than the maximum. If the control is empty, this flag will not be set.

ERROR_PRECISION_EXCEEDED

The precision of the value of a number field is greater than the allowed

precision, and the UA will not be rounding the number for submission. Zero and empty values can never cause this flag to be set.

ERROR_TOO_LONG

The value of a field with a `maxlength` attribute is longer than the attribute allows.

ERROR_PATTERN_MISMATCH

The value of the field with a `pattern` attribute doesn't match the pattern. If the control is empty, this flag will not be set.

ERROR_REQUIRED

The field has the `required` attribute set but has no value.

ERROR_USER_DEFINED

The field was marked invalid from script. See the definition of the `setValidity()` method.

When the definitions above refer to elements that have an attribute set on them, they do not refer to elements on which that attribute is defined not to apply. For example, the `ERROR_REQUIRED` code cannot be set on an `<input type="checkbox">` element, even if that element has the `required` attribute set, since `required` doesn't apply to check boxes.

The **`validate()`** method, present on several of the form control interfaces, causes an `invalid` event to be fired on that control, unless the `validity` of the control is zero. It returns true if the `validity` of the control is zero, otherwise it returns false. ***Recall that this is automatically done during form submission.***

The **`setValidity()`** method sets and resets the `ERROR_USER_DEFINED` bit on the `validity` attribute based on the method's argument. If the argument is false, the bit is set (indicating that the control is not valid), and if the argument is true, the bit is reset (the control is valid). Even attributes that are empty and not required can be marked invalid like this, and would abort form submission if so marked. Setting this bit is persistent, in that the bit remains set until specifically unset using the same method. For instance resetting the form, changing the control value, or moving the element around the document do not affect the value of this bit.

7.5. New DOM attributes for new content attributes

The new `pattern`, `required`, `autocomplete`, `inputmode`, `min`, `max`, `wrap`, `disabled`, `action`, `enctype`, `method` and `replace` attributes simply reflect the current value of their relevant content attribute.

The `type` attribute on the `HTMLButtonElement` interface is changed from read-only to read-write.

The `form` attribute on most of the control interfaces is read-only, but reflects the curret state of the `form` content attribute. If the content attribute is changed, e.g. via the `setAttribute()` method, the DOM attribute should change with it.

7.6. Labels

Form controls all have a **`labels`** DOM attribute that lists all the `label` elements that refer to the control (either through the `for` attribute or via containership), in document order.

Similarly, `HTMLLabelElement`s have a **`control`** DOM attribute that points to the associated element node, if any.

A label must be listed in the `labels` list of the control to which its `control` attribute points, and no other.

The **`changed()`** and **`formchanged()`** methods cause `change` and `formchange` events to be fired on the element. They are intended primarily to be used from `oninput` and `onforminput` handlers to avoid code duplication:

```
<input oninput="changed" onchange="some long algorithm">
```

7.7. Repetition interfaces

The **`RepetitionElement`** interface must be implemented by all elements.

If the element is a repetition template, its **`repetitionType`** DOM attribute must return `REPETITION_TEMPLATE`. Otherwise, if the element is a repetition block, it must return `REPETITION_BLOCK`. Otherwise, it is a normal element, and that attribute should return `REPETITION_NONE`.

Setting `repetitionType` modifies the `repeat` attribute. The `repeat` attribute's namespace depends on the element node's namespace; if the element is in the `http://www.w3.org/1999/xhtml` namespace then the attribute has no namespace, otherwise the *attribute* is in that namespace. If `repetitionType` is set to `REPETITION_NONE`, the attribute is removed. If it is set to `REPETITION_TEMPLATE`, the attribute is set to "`template`". If `repetitionType` is set to `REPETITION_BLOCK`, the `repeat` content attribute is set to the value of the `repetitionIndex` DOM attribute.

The **`repetitionIndex`** attribute must return the current value of the index of the repetition template or block. If the element is a repetition block, setting this attribute must update the `repeat` attribute appropriately (and changing the

attribute directly must affect the value of the `repetitionIndex` attribute). Otherwise, if the element is a repetition template, setting this attribute changes the template's index but does not affect any other aspect of the DOM. If the element is a normal element, it must always return zero, and setting the attribute must have no effect.

The **`repetitionTemplate`** attribute is null unless the element is a repetition block, in which case it points to the block's template. If the block is an orphan repetition block then it returns null.

The **`repetitionBlocks`** attribute is null unless the element is a repetition template, in which case it points to a list of elements (an `HTMLCollection`, although the name of that interface is a misnomer since there is nothing HTML-specific about it). The list consists of all the repetition blocks that have this element as their template. The list is live.

The `addRepetitionBlock()`, `addRepetitionBlockByIndex()`, `moveRepetitionBlock()` and `removeRepetitionBlock()` methods are defined in the section on the repetition model.

The `template` DOM attribute on the `HTMLInputElement` and `HTMLButtonElement` interfaces represents the repetition template that the `template` content attribute refers to. If the content attribute points to a non-existent element or an element that is not a repetition template, the DOM attribute returns null. This DOM attribute is readonly in this version of this specification.

7.8. Loading remote documents

The `FormImplementation` interface can be obtained using binding-specific casting methods on the `implementation` object.

The **`load()`** method on the `FormImplementation` interface returns a `Document` interface, and then queues the specified resource to be loaded into that document. When the document has finished loading, a `load` event fires on the object. If a failure occurs during loading, an `error` event fires instead. The `load` and `error` arguments to the `load` method can be used to specify event handlers for those events.

The **`loadWithBody()`** method is analogous, but allows the author to specify the entity body of the request as a string (e.g. for scripted HTTP POST requests).

The **`loadWithDocument()`** method is analogous, but allows the author to specify the entity body of the request as a `Document` object.

The arguments for these methods have the following meanings:

action

The URI to lead. If a null value is passed, a `TYPE_MISMATCH_ERR` exception is raised.

method

The method (e.g. for HTTP actions, 'POST', 'PUT') used to load the URI. If a null value is passed to the method instead, the default value `post` is assumed.

enctype

The Content-Type of the content. For HTTP requests with entity bodies, this is the value of the submitted Content-Type header. If a null value is passed to the method instead, the default value is `application/x-www-form-urlencoded` for `loadWithBody` and `application/xml` for `loadWithDocument`.

content

The entity body of submission requests. If a null value is passed, a `TYPE_MISMATCH_ERR` exception is raised.

document

The document to submit. If a null value is passed, a `TYPE_MISMATCH_ERR` exception is raised.

load

An event handler to attach to the returned document for the `{"http://www.w3.org/2001/xml-events", "load"}` event. If a null value is passed, no handler is attached.

error

An event handler to attach to the returned document for the `{"http://www.w3.org/2001/xml-events", "error"}` event. If a null value is passed, no handler is attached.

Implementations may limit which hosts, ports, and schemes can be accessed using this method. For example, it is highly recommended that the SMTP port not be allowed, since otherwise it can be used to relay spam on the behalf of the unwitting user. Similarly, cross-domain scripting restrictions are fully expected to apply.

These methods are asynchronous, and are guaranteed to not finish loading the document or signal an error before the running script either completes or yields to the user (e.g. by calling `window.alert()`). Thus, the following code is guaranteed to hook in the event handlers before the document has either finished loading or signalled an error:

```
var d = document.implementation.loadWithBody('http://example.org/search
          'post', 'application/xml', '<search keywords="test search"/>'
d.addEventListenerNS('http://www.w3.org/2001/xml-events', 'load',
                       function () { alert('loaded!'); },
                       false, null);
d.addEventListenerNS('http://www.w3.org/2001/xml-events', 'error',
                       function () { alert('failed!'); },
                       false, null);
```

The following code is equivalent:

```
document.implementation.loadWithBody('http://example.org/search', null,
           'application/xml', '<search keywords="test search"/>',
           function () { alert('loaded!'); },
           function () { alert('failed!'); });
```

The `loadWithDocument()` method serialises its document node and uses the `application/xml` MIME type. It is otherwise identical to the `loadWithBody()` method. If the serialising fails, the method will raise an `INVALID_STATE_ERR` exception.

All other errors will simply generate `error` events on the returned document object.

The semantics of loading documents using these methods are described in the section on the semantics of `method` and `enctype` attributes.

Document loads queued using these methods, and the subsequent firings of load or error events, must occur whether or not the documents go out of scope of the calling script.

For more control over loading remote documents, see DOM3 Load and Save [DOM3LS].

8. Styling form controls

The CSS working group is expected to develop a language designed, amongst other things, for the advanced styling of form controls. In the meantime, technologies such as [HTC] and [XBL] can be used as guides for what is expected.

UAs, in the absence of such advanced styling information, may render form controls described in this draft as they wish. It is recommended that form controls remain faithful to the look and feel of the system's global user interface, though.

Note that [CSS21] explicitly does not define how CSS applies to form controls.

8.1. Relation to the CSS3 User Interface module

[CSS3UI] introduces a number of pseudo-classes for form controls. Their relationship to the form controls described in this specification is described here.

:enabled

Matches form control elements that do not have the `disabled` attribute set.

:disabled

Matches form control elements that do have the `disabled` attribute set.

:checked

Matches radio and check box form control elements that are `checked`.

:indeterminate

Matches no HTML form control elements.

:default

Matches the button (if any) that will be selected if the user presses the enter key (or some equivalent behaviour on less typical systems).

:valid

Matches form control elements that would not have the `invalid` event fired at them if the form was submitted.

:invalid

Matches form control elements that would have the `invalid` event fired at them if the form was submitted.

:in-range

Matches numeric, date-related, or time-related form control elements when the current value is type-correct, greater than or equal to the minimum (if any), and less than or equal to the maximum (if any).

:out-of-range

Matches numeric, date-related, or time-related form control elements when the current value is type-correct, but either less than the minimum or greater than the maximum.

:required

Matches form control elements that have the `required` attribute set.

:optional

Matches form control elements that do not have the `required` attribute set.

:read-only

Matches form control elements that have the `readonly` attribute set.

:read-write

Matches form control elements that do not have the `readonly` attribute set (including `password` fields, although technically they should be called "writeonly").

When the definitions above refer to elements that have an attribute set on them, they do not refer to elements on which that attribute is defined not to apply. For example, the `:read-only` attribute cannot apply to a `<input type="radio">` element, even if that element has the `readonly` attribute set, since `readonly` doesn't apply to radio buttons.

A. XHTML module definition

The Forms Extensions Module provides all of the forms features found in HTML 4.0, plus the extensions described above. Specifically, the Forms Extensions Module supports:

Elements	Attributes	Minimal Content Model
form	Common, accept (ContentTypes), accept-charset (Charsets), action (URI), enctype (ContentType), method ("get"* | "post" | "put" | "delete"), replace ("document"* | "values")	(Heading | List | Block)*
input	Common, accept (ContentTypes), accesskey (Character), action (URI), alt (Text), autocomplete ("on"* | "off"), checked ("checked"), disabled ("disabled"), enctype (ContentType), form (IDREF), help (URI), inputmode (CDATA), maxlength (Number), method ("get" | "post" | "put" | "delete"), min (CDATA), max (CDATA), name (CDATA), pattern (CDATA), precision (CDATA), readonly ("readonly"), replace ("document" | "values") required ("required"), size (Number), src (URI), tabindex (Number), template (IDREF), type ("text"* | "password" | "checkbox" | "radio" | "button" | "submit" | "reset" | "add" | "remove" | "file" | "hidden" | "image" | "datetime" | "date" | "expdate" | "week" | "time" | "number" | "email" | "tel" | "uri"), value (CDATA),	EMPTY
select	Common, autocomplete ("on"* | "off"), disabled ("disabled"), form (IDREF), help (URI), inputmode (CDATA), maxlength (Number), multiple ("multiple"),	(optgroup | option)*

Elements	Attributes	Minimal Content Model
	name (CDATA), pattern (CDATA), required ("required"), size (Number), tabindex (Number)	
optgroup	Common, disabled ("disabled"), label* (Text)	option*
option	Common, disabled ("disabled"), label (Text), selected ("selected"), value (CDATA)	PCDATA
textarea	Common, accesskey (Character), autocomplete ("on"* | "off"), cols (Number), disabled ("disabled"), form (IDREF), help (URI), inputmode (CDATA), maxlength (Number), name (CDATA), readonly ("readonly"), required ("required"), rows (Number), tabindex (Number), wrap ("soft"* | "hard")	PCDATA
output	Common, form (IDREF), name (CDATA)	PCDATA
button	Common, accesskey (Character), action (URI), disabled ("disabled"), enctype (ContentType), form (IDREF), help (URI), method ("get" | "post" | "put" | "delete"), name (CDATA), replace ("document" | "values") tabindex (Number), template (IDREF), type ("button" | "submit"* | "reset" | "add" | "remove"), value (CDATA)	(PCDATA | Heading | List | Block - Form | Inline - Formctrl)*
fieldset	Common, disabled ("disabled"), form (IDREF), help (URI),	(PCDATA | legend | Flow)*
legend	Common, accesskey (Character)	(PCDATA | Inline)*
label	Common, accesskey (Character), for (IDREF)	(PCDATA | Inline - label)*

This module defines two content sets:

Form

form | fieldset

Formctrl

input | select | textarea | output | button | label

When this module is used, it adds the `Form` content set to the `Block` content set and it adds the `Formctrl` content set to the `Inline` content set as these are defined in the Text Module.

All XHTML elements (all elements in the `http://www.w3.org/1999/xhtml` namespace) may have the `repeat` attribute specified. Similarly, the global attribute `repeat` in the `http://www.w3.org/1999/xhtml` namespace may be

specified on any non-XHTML element. These two attributes may either have the value "`template`" or be an integer (an optional '-' character followed by one or more decimal digits).

The `form` element may be placed inside XHTML `head` elements when it is empty.

The `repeat` element in the XHTML namespace is allowed anywhere. It is only allowed to have one attribute, `index`, whose value must be an integer (an optional '-' character followed by one or more decimal digits).

Warning. Documents using the new repetition model with the index substitution feature on ID attributes cannot validate, as the "[" and "]" characters are not valid in IDs.

The `oninput` attribute is added to all the elements that have an `onchange` attribute in the XHTML Intrinsic Events module. The `onformchange`, `onforminput` and `oninvalid` attributes are added to all form control elements (including `fieldset`).

When frames are also allowed, the `target` attribute is added to the `input` and `button` elements.

The Forms Extensions Module is a superset of the Forms and Basic Forms modules. These modules may not be used together in a single document type. Note that the content models in this module differ from those of the XHTML1 Forms module in some subtle ways (for example, the `select` element may be empty).

B. Attribute summary

The `input` element takes a large number of attributes that do not always apply. The following table summarizes which attributes apply to which input types.

type	text	password	checkbox	radio	button	submit	reset	add	remove move-up move-down
accept	-	-	-	-	-	-	-	-	-
accesskey	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
action	-	-	-	-	-	Yes	-	-	-
alt	-	-	-	-	-	-	-	-	-
autocomplete	Yes	Yes	Yes	Yes	-	-	-	-	-
checked	-	-	Yes	Yes	-	-	-	-	-

type	text	password	checkbox	radio	button	submit	reset	add	remove move-up move-down
disabled	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
enctype	-	-	-	-	-	Yes	-	-	-
form	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
help	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
inputmode	Yes	Yes	-	-	-	-	-	-	-
maxlength	Yes	Yes	-	-	-	-	-	-	-
method	-	-	-	-	-	Yes	-	-	-
min	-	-	-	-	-	-	-	-	-
max	-	-	-	-	-	-	-	-	-
name	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
pattern	Yes	Yes	-	-	-	-	-	-	-
precision	-	-	-	-	-	-	-	-	-
readonly	Yes	Yes	-	-	-	-	-	-	-
replace	-	-	-	-	-	Yes	-	-	-
required	Yes	Yes	-	Yes	-	-	-	-	-
size	Yes	Yes	-	-	-	-	-	-	-
src	-	-	-	-	-	-	-	-	-
tabindex	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
target	-	-	-	-	-	Yes	-	-	-
template	-	-	-	-	-	-	-	Yes	-
value	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

References

[CGI]

The CGI Specification. NCSA HTTPd Development Team, November 1995. The CGI Specification is available at http://hoohoo.ncsa.uiuc.edu/cgi/interface.html

[CSJSR]

Client-Side JavaScript Reference (1.3). Netscape Communications Corporation, May 1999. The Client-Side JavaScript Reference (1.3) is available at http://devedge.netscape.com/library/manuals/2000/javascript/1.3/reference/index.html

[CHARMOD]

Character Model for the World Wide Web 1.0, M. Dürst, F. Yergeau, R. Ishida, M. Wolf, T. Texin. W3C, August 2003. The latest version of the Character Model specification is available at http://www.w3.org/TR/charmod/

[CSS21]

CSS 2.1 Specification, B. Bos, T. Çelik, I. Hickson, H. Lie. W3C, September 2003. The latest version of the CSS 2.1 specification is available at http://www.w3.org/TR/CSS21

[CSS3UI]

CSS3 Basic User Interface Module, T. Çelik. W3C, July 2003. The latest version of the CSS3 UI module is available at http://www.w3.org/TR/css3-ui

[CSS3CONTENT]

CSS3 Generated and Replaced Content Module, I. Hickson. W3C, May 2003. The latest version of the CSS3 Generated and Replaced Content module is available at http://www.w3.org/TR/css3-content

[DOM3CORE]

Document Object Model (DOM) Level 3 Core Specification, A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, S. Byrne. W3C, November 2003. The latest version of the DOM Level 3 Core specification is available at http://www.w3.org/TR/DOM-Level-3-Core/

[DOM3EVENTS]

Document Object Model (DOM) Level 3 Events Specification, P. Le Hégaret, T. Pixley. W3C, November 2003. The latest version of the DOM Level 3 Events specification is available at http://www.w3.org/TR/DOM-Level-3-Events/

[DOM2HTML]

Document Object Model (DOM) Level 2 HTML Specification, J. Stenback, P. Le Hégaret, A. Le Hors. W3C, January 2003. The latest version of the DOM Level 2 HTML specification is available at http://www.w3.org/TR/DOM-Level-2-HTML/

[DOM3LS]

Document Object Model (DOM) Level 3 Load and Save Specification, J. Stenback, A. Heninger. W3C, November 2003. The latest version of the DOM Level 3 Load and Save specification is available at http://www.w3.org/TR/DOM-Level-3-LS/

[ECMA262]

ECMAScript Language Specification, Third Edition. ECMA, December 1999. This version of the ECMAScript Language is available at http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM

[HTC]

HTML Components, Chris Wilson. Microsoft, September 1998. The HTML Components submission is available at http://www.w3.org/TR/1998/NOTE-HTMLComponents-19981023

[HTML4]

HTML 4.01 Specification, D. Raggett, A. Le Hors, I. Jacobs. W3C, December 1999. The latest version of the HTML4 specification is available at http://www.w3.org/TR/html4

[ISO8601]

ISO8601:2000 Data elements and interchange formats -- Information interchange -- Representation of dates and times. ISO, December 2000. ISO8601 is available for purchase at http://www.iso.ch/

[RFC959]

File Transfer Protocol (FTP), J. Postel, J. Reynolds. IETF, October 1985. RFC959 is available at http://www.ietf.org/rfc/rfc959

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels, S. Bradner. IETF, March 1997. RFC2119 is available at http://www.ietf.org/rfc/rfc2119

[RFC1738]

Uniform Resource Locators (URL), T. Berners-Lee, L. Masinter, M. McCahill. IETF, Decembed 1998. RFC1738 is available at http://www.ietf.org/rfc/rfc1738

[RFC2368]

The mailto URL scheme, P. Hoffman, L. Masinter, J. Zawinski. IETF, July 1998. RFC2368 is available at http://www.ietf.org/rfc/rfc2368

[RFC2387]

The "data" URL scheme, L. Masinter. IETF, August 1998. RFC2387 is available at http://www.ietf.org/rfc/rfc2387

[RFC2396]

Uniform Resource Identifiers (URI): Generic Syntax, T. Berners-Lee, R. Fielding, L. Masinter. IETF, August 1998. RFC2396 is available at http://www.ietf.org/rfc/rfc2396

[RFC2616]

Hypertext Transfer Protocol -- HTTP/1.1, R. Fielding, J. Gettys, J. Mogul,

H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. IETF, June 1999. RFC2616 is available at http://www.ietf.org/rfc/rfc2616

[RFC2806]

URLs for Telephone Calls, A. Vaha-Sipila. IETF, April 2000. RFC2806 is available at http://www.ietf.org/rfc/rfc2806

[RFC2822]

Internet Message Format, P. Resnick. IETF, April 2001. RFC2822 is available at http://www.ietf.org/rfc/rfc2822

[RFC3023]

XML Media Types, M. Murata, S. St.Laurent, D. Kohn. IETF, January 2001. RFC 3023 is available at http://www.ietf.org/rfc/rfc3023

[RFC3106]

ECML v1.1: Field Specifications for E-Commerce, D. Eastlake, T Goldstein. IETF, April 2001. RFC 3106 is available at http://www.ietf.org/rfc/rfc3106

[XBL]

XML Binding Language, David Hyatt. Mozilla, February 2001. The XBL submission is available at http://www.w3.org/TR/2001/NOTE-xbl-20010223/

[XML]

Extensible Markup Language (XML) 1.0 (Second Edition), T Bray, J Paoli, C. M. Sperberg-McQueen, E. Maler. W3C, October 2000. The latest version of the XML specification is available at http://www.w3.org/TR/REC-xml/

[XHTML1]

XHTML™ 1.1 - Module-based XHTML, M. Altheim, S. McCarron. W3C, May 2001. The latest version of the XHTML 1.1 specification is available at http://www.w3.org/TR/xhtml11

[XForms]

XForms 1.0, M. Dubinko, L. Klotz, R. Merrick, T. Raman. W3C, October 2003. The latest version of the XForms specification is available at http://www.w3.org/TR/xforms

Acknowledgements

Thanks to Håkon Wium Lie, Malcolm Rowe, Maciej Stachowiak, David Hyatt,

Peter N Stark, Christopher Aillon, Jason Kersey, Neil Rashbrook, Brendan Eich, Bert Bos, Jukka K. Korpela, Sebastian Schnitzenbaumer, Daniel Bratell, Jens Lindström, Daniel Brooks, Christian Schmidt, Olli Pettay, Andy Heydon, Susan Borgrink, Martin Honnen, Jonas Sicking, Simon Montagu, Christian Biesinger, David Matja, and John Keiser for their substantial comments.

Thanks also to Rich Doughty, Anne van Kesteren, Alexander J. Vincent, David E. Cleary, Martijn Wargers, Matthew Mastracci, Mark Birbeck, Michael Daskalov, Subramanian Peruvemba, Peter Stark, Shanti Rao, Martin Kutschker, and Rigo Wenning for their comments, to the Slashdot community for some ideas, and to the #mozilla crew, the #opera crew, and the #mrt crew for their ideas and support.

Thanks to the XForms working group for unintentionally giving the incentive to develop this specification.