Media Capture and Streams

W3C Editor's Draft 04 October 2013

This version:
http://dev.w3.org/2011/webrtc/editor/getusermedia.html
Latest published version:
http://www.w3.org/TR/mediacapture-streams/
Latest editor's draft:
http://dev.w3.org/2011/webrtc/editor/getusermedia.html
Previous editor's draft:
http://dev.w3.org/2011/webrtc/editor/archives/20130514/getusermedia.html
Editors:
Daniel C. Burnett, Voxeo
Adam Bergkvist, Ericsson
Cullen Jennings, Cisco
Anant Narayanan, Mozilla (until November 2012)

Abstract

Status of This Document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This document was published by the Web Real-Time Communication Working Group and Device APIs Working Group as an Editor's Draft. If you wish to make comments regarding this document, please send them to public-media-capture@w3.org (subscribe, archives). All comments are welcome.

Publication as an Editor's Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures (Web Real-Time Communication Working

Group, Device APIs Working Group) made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

1. Constrainable Interface

The Constrainable interface allows its consumers to inspect and adjust the properties of the object that implements it. It is broken out as a separate partial interface so that it can be used in other specifications. The core concept is that of a Capability, which consists of a property or feature of an object and the set of its possible values, which may be specified either as a range or as an enumeration. For example, a camera might be capable of framerates (a property) between 20 and 50 frames per second (a range) and may be able to be positioned (a property) facing towards the user, away from the user, or to the left or right of the user (an enumerated set.) The application can query for a Constrainable object's set of Capabilities via the `capabilities()` method.

The application can select the (range of) values it wants for a Capability by means of a Constraint and the `applyConstraints()` method. A Constraint consists of the name of the property, plus the desired value (or a range of desired values.) For example, the application may set a Constraint stating that the framerate of a camera be between 30 and 40 frames per second (a range) or that it wants the camera to be facing the user (a specific value). Constraints can be mandatory or optional. In the case of optional Constraints, the UA MUST consider the Constraints in the order in which they are specified, and MUST try to satisfy each one, but MAY ignore a Constraint if it cannot satisfy it. In the case of mandatory Constraints, the MUST try to satisfy all of them, and MUST call the `errorCallback` if it cannot do so. For example, suppose that an application applies three Constraints, stating that the video aspect ratio should be 3 to 2 (height to width), that the height should be 600 and that the width should be 500. Since these constraints interact with each other (the aspect ratio affects the possible values for height and width, and vice-versa) it is impossible to satisfy all three constraints, so if all the Constraints are mandatory, the UA will call the `errorCallback`. However if any one of the Constraints is optional, the other two can be satisified, so the UA will satisfy the two mandatory ones,

silently ignore the optional one, and call the `successCallback`.

The ordering of optional constraints is significant. In the example in the previous paragraph, suppose that aspect ratio constraint is mandatory and that the height and width constraints are optional. If the height constraint is specified first, then it will be satisfied and the width constraint will be ignored. Thus the height will be set to 600 and the the width will be set to 400. On the other hand, if width is specified before height, the width constraint will be satisfied and the height constraint will be ignored, resulting in width of 500 and height of 750. (Note that the mandatory aspect ratio constraint is enforced in both cases.) The UA will attempt to satisfy as many optional constraints as it can, even if some of them cannot be satisfied and must therefore be ignored. Application authors can therefore implement a backoff strategy by specifying multiple optional constraints for the same property. For example, an application might specify three optional constraints, the first asking for a framerate greater than 500, the second asking for a framerate greater than 400, and the third asking for one greater than 300. If the UA is capable of setting a framerate greater than 500, it will (and the subsequent two constraints will be trivially satisifed.) However, if the UA cannot set the framerate above 500, it will ignore that constraint and attempt to set the framerate above 400. If that fails, it will then try to set it above 300. If the UA cannot satisfy any of the three constraints, it will set the framerate to any value it can get. If the developer wanted to insist on 300 as a lower bound, he could make that a mandatory constraint. In that case, the UA would fail altogether if it couldn't get a value over 300, but would choose a value over 500 if possible, then try for a value over 400.

An application may inspect the set of Constraints currently in effect via the `constraints()` method.

The specific value that the UA chooses for a Capability is referred to as a Setting. For example, if the application applies a Constraint that the framerate must be at least 30 frames per second, and no greater than 40, the Setting can be any intermediate value, e.g., 32, 35, or 37 frames per second. The application can query the current settings of the object's Capabilities via the `settings()` method.

1.1 Interface Definition

```
WebIDL

interface Constrainable {
    Constraints? constraints ();
    void         applyConstraints (Constraints constraints,
ConstraintSuccessCallback successCallback,
ConstraintErrorCallback errorCallback);
                 attribute EventHandler onoverconstrained;
    Settings     settings ();
    Capabilities capabilities ();
};
```

1.1.1 Attributes

onoverconstrained of type EventHandler,
: This event handler, of type overconstrained, MUST be supported by all objects implementing the **Constrainable** interface. The UA MUST raise the `overconstrained` event if changing circumstances at runtime result in the currently valid mandatory constraints no longer being satisfied. The conditions under which this might happen are platform- and application-specific. For example, the user might physically manipulate a camera in a way that made it impossible to provide a resolution that satisified the constraints.

1.1.2 Methods

applyConstraints

The algorithm for applying constraints is stated below. Here are some preliminary definitions that are used in the statement of the algorithm:

- A sequence of values for the properties of an object O satisfy constraint set C if each value a) is in the set of supported values specified by the corresponding Capability of O, and b) is in the set specified by any constraints in C that apply to that property, and c) there is no constraint in C that does not correspond to a Capability in O. (Note that although this definition ignores the difference between mandatory and optional constraints, the algorithm below distinguishes between them.)
- A set of constraints C can be satisifed by an object O if it is possible to choose a sequence of values for the properties of O that satisfy C.
- To apply a set of constraints C to object O is to choose such a sequence of values that satisfy C and assign them as the settings for the properties of O.

When `applyConstraints` is called, the UA MUST queue a task to run the following steps:

1. let *desiredConstraints* be the argument to this function. Each constraint MUST specify one or more values (or a range of values) for its property. A property MAY appear more than once in the list of optional constraints.
2. let *existingConstraints* be the set of constraints currently in effect. (Note that it may be empty.)
3. Let *newConstraints* be an initially empty set of constraints
4. Let *object* be the Constrainable object on which this method was called. Let *copy* be an unconstrained copy of *object* (i.e., *copy* should behave as if it were *object* with *existingConstraints* removed.)
5. If the mandatory constraints in *desiredConstraints* is cannot be satisifed by *copy*, call the `errorCallback`, passing it a list of the mandatory constraints that could not be satisfied, and return. (Note that there may be more than one way of selecting the set of constraints that were not satisfied.) `existingConstraints` remain in effect on *object* in this case.
6. Otherwise add the mandatory constraints to *newConstraints*
7. Iterate over the optional constraints in *desiredConstraints* in the order in which they were specified. For each constraint,if it and *newConstraints* together can be satisfied by *copy*, add it to *newConstraints*. Otherwise, ignore it.
8. In a single operation, remove *oldConstraints* from *object*, apply *newConstraints*, and fire the `successCallback` passing it *newConstraints* as its argument. Note: the UA MAY modify the values of one or more properties of *object* at any time, as long as the resulting set of values satisfy the current set of constraints.

Parameter	Type	Nullable	Optional	Description
constraints	**Constraints**	✗	✗	A new constraint structure to apply to this object.
successCallback	**ConstraintSuccessCallback**	✗	✗	Called if all mandatory constraints can be satisfied.

errorCallback	`ConstraintErrorCallback`	✗	✗	Called if one or more mandatory constraints cannot be satisified.

Return type: `void`

`capabilities`

Returns a sequence of the capabilities that the object supports. Note that it is possible that the underlying hardware may not exactly map to the range defined in the specification. In this case, an implementation SHOULD make a reasonable attempt to translate and scale the hardware's setting onto the mapping provided by the relevant specification.

> ***NOTE***
> ***An example of the user agent providing an alternative mapping: if a source supports a hypothetical fluxCapacitance capability that is defined to be the range from -10 (min) to 10 (max), but the source's (hardware setting) for fluxCapacitance only supports values of "off" "medium" and "full", then the user agent should map the range value of -10 to "off", 10 should map to "full", and 0 should map to "medium". Constraints imposing a strict value of 3 will cause the user agent to attempt to set the value of "medium" on the hardware, and return a fluxCapacitance of 0, the closest supported setting. No error event is raised in this scenario.***

> ISSUE 1
> What about the case where the object uses the same units as the specification, but supports only a sub-range of the specified values? What about an object whose fluxCapacitance range is only -5 to 5? Should it map those to -10 to 10, or leave them as they are?

No parameters.
Return type: `Capabilities`

`constraints`

Returns all the `Constraints` that were applied to the object in the last succesfull call of `applyConstraints()`. The UA MUST return only the constraints that were successfully applied, and MUST maintain the order that they were specified in.

If no mandatory constraints have been defined, the `mandatory` field will not be present (it will be undefined). If no optional constraints have been defined, the `optional` field will not be present (it will be undefined). If neither optional, nor mandatory constraints have been created, the value `null` is returned.

Note that modifying the return value of this function does not modify the set of constraints that apply to the object. Only `applyConstraints()` can do that.

No parameters.
Return type: **`Constraints`**, nullable

settings
Returns the current settings of all the properties of the object, whether they are platform defaults or have been set by `applyConstraints()`. Note that the actual setting of a property MUST be a single value.
No parameters.
Return type: `Settings`

1.1.3 applyConstraints Success and Failure Callbacks

1.1.3.1 ConstraintSuccessCallback

```
WebIDL

callback ConstraintSuccessCallback = void (Constraints
successfullConstraints);
```

1.1.3.1.1 Callback `ConstraintSuccessCallback` Parameters

`successfullConstraints` of type `Constraints`
The set of constraints that were successfully applied. Note that this may be a subset of those passed into `applyConstraints` since optional constraints may not have been satisfied.

1.1.3.2 ConstraintError

```
WebIDL

[NoInterfaceObject]
interface ConstraintError {
    readonly    attribute Constraints constraintNames;
};
```

1.1.3.2.1 Attributes

`constraintNames` of type *Constraints*, readonly
Contains the set of mandatory constraints that could not be satisified. The `optional` field in this object will be undefined.

1.1.3.3 ConstraintErrorCallback

```
WebIDL

callback ConstraintErrorCallback = void (ConstraintError error);
```

1.1.3.3.1 Callback `ConstraintErrorCallback` Parameters

`error` of type `ConstraintError`
An object holding the mandatory constraints that could not be satisfied.

1.2 Capabilities, Constraints, and Settings

1.2.1 The Property Registry

For each class/interface that implements Constrainable, there MUST be a corresponding registry that defines the constrainable properties of that class of object. The registry entries MUST contain the name of each property along with its set of legal values. The syntax for the specification of the set of legal values depends on the type of the values. We define three types here: boolean values, min-max ranges, and enumerated lists of values. Boolean values are built into JavaScript, the other two types are defined below:

1.2.1.1 PropertyValueRange and PropertyValueList

```
WebIDL

  [Constructor (DOMString property, any min, any max)]
  interface PropertyValueRange {
                    attribute any max;
                    attribute any min;
  };
```

1.2.1.1.1 Attributes

`max` of type any,
The maximum legal value of this property.

The type of this value is specific to the object and property in question.

`min` of type any,
The minimum value of this Property.

The type of this value is specific to the object and property in question.

```
WebIDL

  typedef sequence<DOMString> PropertyValueList;
```

PropertyValueLists are just an array of supported `DOMString` values.

> ISSUE 2
>
> Should we allow registries to define new or different syntaxes for capabilities? For example could a registry introduce a new Capability that had a min, a max, and a step value?

ISSUE 3

Should we allow the use of multiple registries? For example, a spec might define a registry of Capabilities, and an implementation might define extensions. In that case, we could think of the implementation as having its own registry, and the final set of Capabilities as being the union of the two.

1.2.2 Capabilities

Capabilities a dictionary containing one or more key-value pairs, where each key MUST be a constrainable property defined in the associated registry, and each value SHOULD be a subset of the set of values defined for that property in the registry. The exact syntax of the value expression depends on the type of the property. The Capabilities dictionary specifies the subset of the constrainable properties and values from the registry that the UA supports. Note that a UA MAY support only a subset of the properties that are defined in the registry, and MAY support a subset of the set values for those properties that it does support.

1.2.3 Settings

Settings is a dictionary containing one or more key-value pairs. It MUST contain each key returned in `capabilities()`. There MUST be a single value for each key and the value MUST a member of the set defined for that property by `capabilities()`. Thus the `Settings` dictionary contains the actual values that the UA has chosen for the object's Capabilities.

1.2.4 Constraints

```
WebIDL

dictionary Constraints {
    ConstraintSet? mandatory;
    Constraint[]   optional;
};
```

1.2.4.1 Dictionary `Constraints` Members

`mandatory` of type ConstraintSet, nullable

The set of Constraints that the UA MUST satisfy or else call the `errorCallback`. Note that a given property name may occur only once in this set.

`optional` of type array of Constraint

The set of Constraints that the UA SHOULD try to satisfy but MAY ignore if they cannot be satisified. The order of these constraints is significant. In particular, when they are passed as an argument to `applyConstraints`, the UA MUST try to satisfy them in the order that is specified. Thus if optional constraints C1 and C2 can be satisfied individually, but not together, then whichever of C1 and C2 is first in this list will be satisfied, and the other will not. The UA MUST attempt to satisfy all optional constraints in the list, even if some cannot be satisfied. Thus, in the preceeding example, if optional constraint C3 is specified after C1

and C2, the UA will attempt to satisfy C3 even though C2 cannot be satisified. Note that a given property name may occur multiple times in this set.

Each entry in the ConstraintSet and Constraint dictionaries corresponds to a property and specifies a subset of its legal values. Applying a constraint instructs that UA to restrict the setting of the corresponding Capability to that value or range of values. A given property MAY occur both in the mandatory and the optional constraints list, and MAY occur more than once in the optional constraints list.

1.2.4.2 ConstraintSet

A ConstraintSet is a dictionary containing one or more key-value pairs, where each key MUST be a constrainable property defined in the associated registry, and each value SHOULD be a subset of the set of values defined for that property in the registry. The exact syntax of the value expression depends on the type of the property.

1.2.4.3 Constraint

A Constraint is a dictionary containing exactly one key-value pair, where the key MUST be a constrainable property defined in the associated registry, and the value SHOULD be a subset of the set of values defined for that property in the registry. The exact syntax of the value expression depends on the type of the property.