REVIEW DRAFT

FIDO U2F JavaScript API

FIDO Alliance Review Draft 15 September 2016

This version:
https://fidoalliance.org/specs/fido-u2f-v1.1-rd-20160915/fido-u2f-javascript-api-v1.1-v1.1-rd-20160915.html
Previous version:
https://fidoalliance.org/specs/fido-u2f-javascript-api-v1.1-RD-20140209.html
Editors:
Dirk Balfanz, Google, Inc.
Arnar Birgisson, Google, Inc.
Juan Lang, Google, Inc.

Background

This section is non-normative.

CHANGES: This version *version 1.1* of the FIDO U2F JavaScript API specification supersedes version JavaScript API 1.0. The major difference between these two versions is the way that requests to be signed are formatted between the RP and the client: In version 1.0, a separate *appId* and *challenge* were sent for every *keyHandle*, whereas in version 1.1, an optimization is made that requires only a single *appId* and *challenge* for multiple *keyHandles*.

LOW-LEVEL API: Although this specification refers to two separate API levels, we want to discourage a Relying Party (RP) from implementing directly against the Low-level MessagePort API as this may be deprecated in future versions of this specification. RPs should rather implement against the High-level JavaScript API and use a library that abstracts the lower-level MessagePort API if required.

Abstract

The U2F JavaScript API consists of two calls - one to register a U2F token with a relying party (i.e., cause the U2F token to generate a new key pair, and to introduce the new public key to the relying party), and one to sign an identity assertion (i.e., exercise a previously-registered key pair).

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 FIDO Alliance publications and the latest revision of this technical report can be found in the FIDO Alliance specifications index at https://www.fidoalliance.org/specifications/.

This document was published by the FIDO Alliance as a Review Draft. This document is intended to become a FIDO Alliance Proposed Standard. If you wish to make comments regarding this document, please Contact Us. All comments are welcome.

This is a Review Draft Specification and is not intended to be a basis for any implementations as the Specification may change. Permission is hereby granted to use the Specification solely for the purpose of reviewing the Specification. No rights are granted to prepare derivative works of this Specification. Entities seeking permission to reproduce portions of this Specification for other uses must contact the FIDO Alliance to determine whether an appropriate license for such use is available.

Implementation of certain elements of this Specification may require licenses under third party intellectual property rights, including without limitation, patent rights. The FIDO Alliance, Inc. and its Members and any other contributors to the Specification are not, and shall not be held, responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

THIS FIDO ALLIANCE SPECIFICATION IS PROVIDED “AS IS” AND WITHOUT ANY WARRANTY OF ANY KIND, INCLUDING, WITHOUT LIMITATION, ANY EXPRESS OR IMPLIED WARRANTY OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Table of Contents

1. Notation

Type names, attribute names and element names are written as`code`.

String literals are enclosed in “”, e.g. “UAF-TLV”.

In formulas we use “|” to denote byte wise concatenation operations.

DOM APIs are described using the ECMAScript [ECMA-262] bindings for WebIDL [WebIDL].

U2F specific terminology used in this document is defined in [FIDOGlossary].

1.1 Key Words

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].

Below we explain some of the terms used in this document:

Term	Definition
websafe-base64 encoding	This is the "Base 64 Encoding with URL and Filename Safe Alphabet" from Section 5 in [RFC4648] without padding.
stringified javascript object	This is the JSON object (i.e., a string starting with "{" and ending with "}") whose keys are the property names of the javascript object, and whose values are the corresponding property values. Only "data objects" can be stringified, i.e., only objects whose property names and values are supported in **JSON.**

2. Introduction

Note: Reading the 'FIDO U2F Overview' (see [U2FOverview] in bibliography) is recommended as a background for this document.

A Relying Party (RP) consumes identity assertions from U2F tokens. The RP's web pages communicate with the U2F tokens on the client through a JavaScript API. The RP also needs to perform some verification steps on the server side (see below). How the data obtained by the RP's JavaScript is transferred to the RP's server is out of scope of this document. We instead describe the JavaScript API used by the RP.

3. API Levels

The U2F API may be exposed to web pages on two levels. On the required lower level, RPs interact with the FIDO client through a MessagePort [WEBMESSAGING] object. The low-level MessagePort API defines the message formats for messages sent and received on the port, for the two operations supported by the API. This specification does not describe how such a port is made available to RP web pages, as this is (for now) implementation and browser dependent.

For convenience, the FIDO client may also expose a high-level JavaScript API built on top of the MessagePort API. This API consists of functions corresponding to the different requests that can be made to the FIDO client. These functions respond to the RP asynchronously by invoking a callback.

Why two API levels? The messaging API requires only that pages obtain a MessagePort instance to the FIDO client, i.e. no code needs to be injected to JavaScript context of the RP's pages. This allows RPs to keep full control over the JS running in their pages. The JS API is offered as a convenient abstraction of the messaging API, and is useful for RP developers to quickly integrate U2F into their websites.

3.1 Low-level MessagePort API

RP web pages communicate with the FIDO client over an instance of the HTML5 MessagePort interface. Client implementations may choose how this instance is made available to web pages.

Messages sent to the FIDO client should be `U2fRequest` dictionaries:

WebIDL

```
dictionary U2fRequest {
    DOMString      type;
    DOMString?     appId;
    unsigned long? timeoutSeconds;
    unsigned long? requestId;
};
```

3.1.1 Dictionary `U2fRequest` Members

`type` of type DOMString
: The type of request, either "u2f_register_request" or "u2f_sign_request".

appId of type DOMString, nullable
An application identifer for the request. If none is given, the origin of the calling web page is used.

timeoutSeconds of type unsigned long, nullable
A timeout for the FIDO Client's processing, in seconds.

requestId of type unsigned long, nullable
An integer identifying this request from concurrent requests.

Subtypes of `U2fRequest` for register and sign requests are defined below in their respective sections. If `timeoutSeconds` is omitted, timeout behavior is unspecified. If `requestId` is present, the FIDO client must include its value the corresponding `Response` dictionary under the same key.

Responses from the FIDO client to the RP webpage should be `U2fResponse` dictionaries:

WebIDL

```
dictionary U2fResponse {
    DOMString                                type;
    (Error or RegisterResponse or SignResponse) responseData;
    unsigned long?                           requestId;
};
```

3.1.2 Dictionary `U2fResponse` Members

type of type DOMString
The response type, either "u2f_register_response" or "u2f_sign_response"

responseData of type (Error or RegisterResponse or SignResponse)
The response data, see 5. U2F operations

requestId of type unsigned long, nullable
The `requestId` value of the corresponding request, if present. Otherwise omitted.

Errors are indicated by an `Error` dictionary sent as the response data. An error dictionary can be identified by checking for its non-zero integer `errorCode` key. `RegisterResponse` and `SignResponse` do not define this key. An error object may optionally contain a string `errorMessage` with further description of the error.

WebIDL

```
dictionary Error {
    ErrorCode  errorCode;
    DOMString? errorMessage;
};
```

3.1.3 Dictionary `Error` Members

errorCode of type *ErrorCode*
An error code from the `ErrorCode` enumeration.

errorMessage of type DOMString, nullable
A description of the error.

3.2 High-level JavaScript API

A FIDO client may provide a JavaScript convenience API that abstracts the lower-level MessagePort API. Implementations may choose how to make such an API available to RP web pages. If such an API is provided, it should provide a namespace object `u2f` of the following interface.

WebIDL

```
interface u2f {
    void register (DOMString appId, sequence<RegisterRequest> registerRequests, sequence<RegisteredKey> registeredKeys, function(R
    void sign (DOMString appId, DOMString challenge, sequence<RegisteredKey> registeredKeys, function(SignResponse or Error) callb
};
```

3.2.1 Methods

register

Parameter	Type	Nullable	Optional	Description
appId	`DOMString`	✘	✘	An application id for the request.
registerRequests	`sequence<RegisterRequest>`	✘	✘	Register requests, one for each U2F protocol version accepted by RP
registeredKeys	`sequence<RegisteredKey>`	✘	✘	Identifiers for already registered tokens
callback	`function(RegisterResponse or Error)`	✘	✘	Response handler
opt_timeoutSeconds	`unsigned long`	✔	✔	Timeout in seconds, for the FIDO client's handling of the request.

Return type: `void`

sign

Parameter	Type	Nullable	Optional	Description
appId	`DOMString`	✘	✘	An application id for the request.
challenge	`DOMString`	✘	✘	The websafe-base64-encoded challenge.
registeredKeys	`sequence<RegisteredKey>`	✘	✘	Sign requests, one for each registered token
callback	`function(SignResponse or Error)`	✘	✘	Response handler
opt_timeoutSeconds	`unsigned long`	✔	✔	Timeout in seconds, for the FIDO client's handling of the request.

Return type: `void`

The JavaScript API must invoke the provided callbacks with either response objects, or an error object. An error can be detected by testing for a

non-zero `errorCode` key.

EXAMPLE 1

```
u2f.sign(reqs, function(response) {
                 if (response.errorCode) {
                   // response is an Error
                   ...
                 } else {
                   // response is a SignResponse
                   ...
                 }
               });
```

4. U2F transports

A U2F token may support one or more of the low-level transport mechanisms. In order to improve user experience, the RP may indicate to the client which transports a particular key handle uses. It does so through the use of the `Transport` enumeration:

WebIDL

```
enum Transport {
    "bt",
    "ble",
    "nfc",
    "usb"
};
```

Enumeration description	
`bt`	Bluetooth Classic (Bluetooth BR/EDR)
`ble`	Bluetooth Low Energy (Bluetooth Smart)
`nfc`	Near-Field Communications
`usb`	USB HID

For convenience, all the transports supported by a token may be referred to by:

WebIDL

```
typedef sequence<Transport> Transports;
```

Throughout this specification, the identifier Transports is used to refer to the sequence<`Transport`> type.

5. U2F operations

Regardless of the API level used, the U2F client must support the two operations of registering a token, and generating a signed assertion. This section describes the interface to each operation, their corresponding request and response dictionaries and possible error codes.

5.1 Registration

To register a U2F token for a user account at the RP, the RP must:

- decide which U2F protocol version(s) of device it wants to register,
- pick an appropriate application id for the registration request,
- generate a random challenge, and
- store all private information associated with the registration (expiration times, user ids, etc.)

The RP may choose an application id for the registration request. If none is chosen, the RP's web origin is used as the application id. The new key pair that the U2F token generates will be associated with this application id. (For application id details see [FIDOAppIDAndFacets] in bibliography).

For each version it is willing to register, it then prepares a `RegisterRequest` dictionary as follows:

WebIDL

```
dictionary RegisterRequest {
    DOMString version;
    DOMString challenge;
};
```

5.1.1 Dictionary `RegisterRequest` Members

`version` of type DOMString
: The version of the protocol that the to-be-registered token must speak. E.g. "U2F_V2".

`challenge` of type DOMString
: The websafe-base64-encoded challenge.

Additionally, the RP should prepare a `RegisteredKey` for each U2F token that is already registered for the current user as follows:

WebIDL

```
dictionary RegisteredKey {
    DOMString   version;
    DOMString   keyHandle;
    Transports? transports;
    DOMString?  appId;
};
```

5.1.2 Dictionary `RegisteredKey` Members

`version` of type DOMString
: Version of the protocol that the to-be-registered U2F token must speak. E.g. "U2F_V2"

keyHandle of type DOMString
: The registered keyHandle to use for signing, as a websafe-base64 encoding of the key handle bytes returned by the U2F token during registration.

transports of type *Transports*, nullable
: The transport(s) this token supports, if known by the RP.

appId of type DOMString, nullable
: The application id that the RP would like to assert for this key handle, if it's distinct from the application id for the overall request. (Ordinarily this will be omitted.)

The RP delivers a registration request to the FIDO client either via the low-level MessagePort API, or by invoking the high-level JavaScript API. Using the low-level MessagePort API, the RP would construct a message of the `U2fRegisterRequest` type:

WebIDL

```
dictionary U2fRegisterRequest : U2fRequest {
    DOMString               type = 'u2f_register_request';
    sequence<RegisterRequest> registerRequests;
    sequence<RegisteredKey>   registeredKeys;
};
```

5.1.3 Dictionary `U2fRegisterRequest` Members

type of type DOMString, defaulting to `'u2f_register_request'`
: sequence<RegisterRequest> registerRequests

registerRequests of type sequence<*RegisterRequest*>

registeredKeys of type sequence<*RegisteredKey*>
: An array of RegisteredKeys representing the U2F tokens registered to this user.

EXAMPLE 2

```
// Low-level API
var port = <obtain U2F MessagePort in a browser specific manner>;
port.addEventListener('message', responseHandler);
port.postMessage({
 'type': 'u2f_register_request',
 'appId': <Application id>,
 'registerRequests': [<RegisterRequest instance>, ...],
 'registeredKeys': [<RegisteredKey for known token 1>, ...],
 'timeoutSeconds': 30,
 'requestId': <unique integer>  // optional
});
```

Using the high-level API, the values are passed as parameters:

EXAMPLE 3

```
// High-level API
u2f.register(<Application id>,
             [<RegisterRequest instance>, ...],
             [<RegisteredKey for known token 1>, ...],
             registerResponseHandler);
```

The FIDO client should treat the order of RegisterRequest dictionaries in the first parameter as a prioritized list. That is, if multiple tokens are present that support more than one version provided by the RP, the version that appears first should be selected. Note that this means multiple RegisterRequests with the same version are redundant, since the first one will always be selected.

Note also that the responseHandler in the low-level API receives a `Response` object, while the `registerResponseHandler` in the high-level API receives the `Error` or `RegisterResponse` objects directly.

The FIDO client will create the raw registration messages from this data (see [U2FRawMsgs] in bibliography), and attempt to perform a registration operation with a U2F token. The registration request message is then used to register a U2F token that is not already registered (if such a token is present).

Note that as part of creating the registration request message, the FIDO client will create a Client Data object (see [U2FRawMsgs]). This Client Data object will be returned to the caller as part of the registration response (see below).

If the registration is successful, the FIDO client returns (via the message port, or the JS API callback) a `RegisterResponse` dictionary as follows.

WebIDL

```
dictionary RegisterResponse {
    DOMString version;
    DOMString registrationData;
    DOMString clientData;
};
```

5.1.4 Dictionary `RegisterResponse` Members

version of type DOMString
: The version of the protocol that the registered token speaks. E.g. "U2F_V2".

registrationData of type DOMString
: The raw registration response websafe-base64

clientData of type DOMString
: The client data created by the FIDO client, websafe-base64 encoded.

For the contents of these fields, refer to [U2FRawMsgs] (see bibliography).

5.2 Generating signed identity assertions

To obtain an identity assertion from a locally-attached U2F token, the RP must

- generate a random challenge, and

- prepare a `RegisteredKey` object for each U2F token that the user has currently registered with the RP.

The RP delivers a sign request to the FIDO client either via the low-level MessagePort API, or by invoking the high-level JavaScript API. Using the low-level MessagePort API, the RP would construct a message of the `U2fSignRequest` type:

WebIDL

```
dictionary U2fSignRequest : U2fRequest {
    DOMString               type = 'u2f_sign_request';
    DOMString               challenge;
    sequence<RegisteredKey> registeredKeys;
};
```

5.2.1 Dictionary `U2fSignRequest` Members

`type` of type DOMString, defaulting to `'u2f_sign_request'`
: DOMString challenge

`challenge` of type DOMString
: The websafe-base64-encoded challenge.

`registeredKeys` of type sequence<*RegisteredKey*>
: An array of RegisteredKeys representing the U2F tokens registered to this user.

EXAMPLE 4

```
// Low-level API
var port = <obtain U2F MessagePort in a browser specific manner>;
port.addEventListener('message', responseHandler);
port.postMessage({
 'type': 'u2f_sign_request',
 'appId': <Application id>,
 'challenge': <random challenge>,
 'registeredKeys': [<RegisteredKey for known token 1>, ...],
 'timeoutSeconds': 30,
 'requestId': <unique integer>  // optional
});
```

In response to a sign request, the FIDO client should perform the following steps:

- Verify the application identity of the caller.
- Using the provided challenge, create a client data object.
- Using the client data, the application id, and the key handle, create a raw authentication request message (see [U2FRawMsgs] in bibliography) and send it to the U2F token.

When the RP provides the `transports` value for any `RegisteredKey`, the client may treat that value has a hint about which transports to prefer for the key handle. The client may also use the transports as a hint about user interface, if the client presents any. Irrespective of whether the RP sets any `transports` value for any `RegisteredKey`, the client should send each key handle over all transports supported by the client.

Eventually the FIDO client must respond (via the MessageChannel or the provided callback). In the case of an error, an `Error` dictionary is returned. In case of success, a `SignResponse` is returned.

WebIDL

```
dictionary SignResponse {
    DOMString keyHandle;
    DOMString signatureData;
    DOMString clientData;
};
```

5.2.2 Dictionary `SignResponse` Members

`keyHandle` of type DOMString
: The keyHandle of the RegisteredKey that was processed.

`signatureData` of type DOMString
: The raw response from U2F device, websafe-base64 encoded.

`clientData` of type DOMString
: The client data created by the FIDO client, websafe-base64 encoded.

If there are multiple U2F tokens that responded to the authentication request, the FIDO client will pick one of the responses and pass it to the caller.

5.3 Error codes

When an `Error` object is returned, its `errorCode` field is set to a non-negative integer indicating the general error that occurred, from the following enumeration.

WebIDL

```
interface ErrorCode {
    const short OK = 0;
    const short OTHER_ERROR = 1;
    const short BAD_REQUEST = 2;
    const short CONFIGURATION_UNSUPPORTED = 3;
    const short DEVICE_INELIGIBLE = 4;
    const short TIMEOUT = 5;
};
```

5.3.1 Constants

`OK` of type short
: Success. Not used in errors but reserved

`OTHER_ERROR` of type short

An error otherwise not enumerated here

`BAD_REQUEST` of type short
The request cannot be processed

`CONFIGURATION_UNSUPPORTED` of type short
Client configuration is not supported

`DEVICE_INELIGIBLE` of type short
The presented device is not eligible for this request. For a registration request this may mean that the token is already registered, and for a sign request it may mean the token does not know the presented key handle.

`TIMEOUT` of type short
Timeout reached before request could be satisfied

5.4 Backward compatibility with U2F 1.0 API

For backward compatibility with the U2F 1.0 API, the RP may prepare a `SignRequest` in lieu of a `RegisteredKey` for each U2F token that is already registered for the current user. See JavaScript API 1.0 for the specification of `SignRequest`.

Similarly, U2F clients may implement backward compatibility with version 1.0 by accepting a `signRequests` key in lieu of `registeredKeys`.

A. References

A.1 Normative references

[ECMA-262]
ECMAScript Language Specification. URL: https://tc39.github.io/ecma262/

[FIDOAppIDAndFacets]
D. Balfanz, B. Hill, R. Lindemann, D. Baghdasaryan, *FIDO AppID and Facets v1.0*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-appid-and-facets-v1.1-rd-20160915.html
PDF: fido-appid-and-facets-v1.1-rd-20160915.pdf

[FIDOGlossary]
R. Lindemann, D. Baghdasaryan, B. Hill, J. Hodges, *FIDO Technical Glossary*. FIDO Alliance Proposed Standard. URLs:
HTML: fido-glossary-v1.1-rd-20160915.html
PDF: fido-glossary-v1.1-rd-20160915.pdf

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels* March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC4648]
S. Josefsson, *The Base16, Base32, and Base64 Data Encodings (RFC 4648)*, IETF, October 2006, URL: http://www.ietf.org/rfc/rfc4648.txt

[U2FRawMsgs]
D. Balfanz, *FIDO U2F Raw Message Formats v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: ./fido-u2f-raw-message-formats-v1.1-rd-20160915.pdf

[WEBMESSAGING]
Ian Hickson. *HTML5 Web Messaging*. 19 May 2015. W3C Recommendation. URL: https://www.w3.org/TR/webmessaging/

[WebIDL]
Cameron McCormack; Boris Zbarsky. *WebIDL Level 1*. 15 September 2016. W3C Proposed Recommendation. URL: https://www.w3.org/TR/WebIDL-1/

A.2 Informative references

[U2FOverview]
S. Srinivas, D. Balfanz, E. Tiffany, *FIDO U2F Overview v1.0*. FIDO Alliance Review Draft (Work in progress.) URL: ./fido-u2f-overview-v1.1-rd-20160915.pdf