FIDO 2.0: Web API for accessing FIDO 2.0 credentials

W3C Member Submission 20 November 2015

This version:
http://www.w3.org/Submission/2015/SUBM-fido-web-api-20151120/
Latest published version:
http://www.w3.org/Submission/fido-web-api/
Editor:
Vijay Bharadwaj, Microsoft, vijay.bharadwaj@microsoft.com
Authors:
Hubert Le Van Gong, PayPal, hlevangong@paypal.com
Dirk Balfanz, Google, balfanz@google.com
Alexei Czeskis, Google, aczeskis@google.com
Arnar Birgisson, Google, arnarb@google.com
Jeff Hodges, PayPal, Jeff.Hodges@paypal.com

Abstract

This specification defines an API that enables web pages to access FIDO 2.0 compliant strong cryptographic credentials through browser script. Conceptually, credentials are stored on a FIDO 2.0 authenticator, and each credential is bound to a single Relying Party. Authenticators are responsible for ensuring that no operation is performed without the user's consent. The user agent mediates access to credentials in order to preserve user privacy.

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 can be found in the W3C technical reports index at http://www.w3.org/TR/.

By publishing this document, W3C acknowledges that the Submitting Members have made a formal Submission request to W3C for discussion. Publication of this document by W3C

indicates no endorsement of its content by W3C, nor that W3C has, is, or will be allocating any resources to the issues addressed by it. This document is not the product of a chartered W3C group, but is published as potential input to the W3C Process. A W3C Team Comment has been published in conjunction with this Member Submission. Publication of acknowledged Member Submissions at the W3C site is one of the benefits of W3C Membership. Please consult the requirements associated with Member Submissions of section 3.3 of the W3C Patent Policy. Please consult the complete list of acknowledged W3C Member Submissions.

Table of Contents

1. Use Cases

This section is non-normative.

This specification defines an API for web pages to access FIDO 2.0 credentials through JavaScript, for the purpose of strongly authenticating a user. FIDO 2.0 credentials are always bound to a single FIDO Relying Party, and the API respects this requirement. Credentials created by a Relying Party can only be accessed by web origins belonging to that Relying Party. Additionally, privacy across Relying Parties must be maintained; scripts must not be able to detect any properties, or even the existence, of credentials belonging to other Relying Parties.

FIDO 2.0 credentials are located on authenticators, which can use them to perform operations subject to user consent. Broadly, authenticators are of two types:

1. *Embedded authenticators* have their operation managed by the same computing device (e.g., smart phone, tablet, desktop PC) as the user agent is running on. For instance, such an authenticator might consist of a Trusted Platform Module (TPM) or Secure Element (SE) integrated into the computing device, along with appropriate platform software to mediate access to this device's functionality.
2. *External authenticators* operate autonomously from the device running the user agent, and accessed over a transport such as Universal Serial Bus (USB), Bluetooth Low Energy (BLE) or Near Field Communications (NFC).

Note that an external authenticator may itself contain an embedded authenticator. For example, consider a smart phone that contains a FIDO 2.0 credential. The credential may be accessed by a web browser running on the phone itself. In this case the module containing the credential is functioning as an embedded authenticator. However, the credential may also be accessed over BLE by a user agent on a nearby laptop. In this latter case, the phone is functioning as an external authenticator. These modes may even be used in a single end-to-end user scenario. One such scenario is described in the remainder of this section.

1.1 Registration (embedded authenticator mode)

- On the phone:
    - User goes to example.com in the browser, and signs in using whatever method they have been using (possibly a pre-FIDO method such as a password).
    - The phone prompts, "Do you want to register this device with example.com?"
    - User agrees.
    - The phone prompts the user for a previously configured authorization gesture (PIN, biometric, etc.); the user provides this.
    - Website shows message, "Registration complete."

1.2 Authentication (external authenticator mode)

- On the laptop:
    - User goes to example.com in browser, sees an option "Sign in with your phone."
    - User chooses this option and gets a message from the browser, "Please complete this action on your phone."

- Next, on the phone:
    - User sees a discreet prompt or notification, "Sign in to example.com."
    - User selects this prompt / notification.
    - User is shown a list of their example.com identities, e.g., "Sign in as Alice / Sign in as Bob."
    - User picks an identity, is prompted for an authorization gesture (PIN, biometric, etc.) and provides this.
- Now, on the laptop:
    - Web page shows that the selected user is signed in, and navigates to the signed-in page.

1.3 Other configurations

A variety of additional use cases and configurations are also possible, including (but not limited to):

- User goes to example.com on their laptop, is guided through a flow to create and register a credential on their phone.
- User employs a FIDO 2.0 credential as described above to authorize a single transaction, such as a payment or other financial transaction.

2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, and *SHOULD* are to be interpreted as described in [*RFC2119*].

This specification defines criteria for a *conforming user agent*. A user agent *MUST* behave as described in this specification in order to be considered conformant. User agents *MAY* implement algorithms given in this specification in any way desired, so long as the end result is indistinguishable from the result that would be obtained by the specification's algorithms. A conforming FIDO Credential API user agent *MUST* also be a conforming implementation of the IDL fragments of this specification, as described in the “Web IDL” specification. [*WebIDL-1*]

2.1 Dependencies

This specification relies on several other underlying specifications.

HTML5
: The concept of *origin* and the `Window` interface are defined in [*HTML5*].

Web IDL
: Many of the interface definitions and all of the IDL in this specification depend on [*WebIDL-1*]. This updated version of the Web IDL standard adds support for *Promises*, which are now the preferred mechanism for asynchronous interaction in all new web APIs.

DOM
: *DOMException* and the DOMException values used in this specification are defined in [*DOM4*].

Web Cryptography API
: The *AlgorithmIdentifier* type and the method for normalizing an algorithm are defined in [*WebCryptoAPI*].

3. FIDO Authenticator model

The API defined in this specification implies a specific abstract functional model for a FIDO authenticator. This section describes the FIDO authenticator model. Client platforms may implement and expose this abstract model in any way desired. However, the behavior of the client's FIDO Credential API implementation, when operating on the embedded and external authenticators supported by that platform, *MUST* be indistinguishable from the behavior specified in the FIDO Credential API section.

In this abstract model, each FIDO authenticator stores some number of FIDO credentials. Each FIDO credential has an identifier which is unique (or extremely unlikely to be duplicated) among all FIDO credentials. Each credential is also associated with a FIDO Relying Party, whose identity is represented by a Relying Party Identifier (RP ID).

A client must connect to a FIDO authenticator in order to invoke any of the operations of that authenticator. This connection defines an authenticator session. A FIDO authenticator must maintain isolation between sessions. It may do this by only allowing one session to exist at any particular time, or by providing more complicated session management.

The following operations can be invoked by the client in an authenticator session.

3.1 The *authenticatorMakeCredential* operation

This operation must be invoked in an authenticator session which has no other operations in progress. It takes the following input parameters:

- The web origin of the script on whose behalf the operation is being initiated, as determined by the user agent and the client.
- The RP ID corresponding to the above web origin, as determined by the user agent and the client.
- All input parameters accepted by the `makeCredential` method, specified below.

When this operation is invoked, the authenticator obtains user consent for creating a new credential. The prompt for obtaining this consent is shown by the authenticator if it has its own output capability, or by the user agent otherwise. Once user consent is obtained, the authenticator generates the appropriate cryptographic keys and creates a new credential. It then associates the credential with the specified RP ID such that it will be able to retrieve the RP ID later, given the credential ID.

On successful completion of this operation, the authenticator returns the type and unique identifier of this new credential to the user agent.

If the user refuses consent, the authenticator returns an appropriate error status to the client.

3.2 The *authenticatorGetAssertion* operation

This operation must be invoked in an authenticator session which has no other operations in progress. It takes the following input parameters:

- The web origin of the script on whose behalf the operation is being initiated, as determined by the user agent and the client.
- The RP ID corresponding to the above web origin, as determined by the user agent and the client.
- All input parameters accepted by the `getAssertion` method, specified below.

When this method is invoked, the authenticator allows the user to select a credential from among the credentials associated with that Relying Party and matching the specified criteria, then obtains user consent for using that credential. The prompt for obtaining this consent may be shown by the authenticator if it has its own output capability, or by the user agent otherwise. Once a credential is selected and user consent is obtained, the authenticator computes a cryptographic signature using the credential's private key and constructs an assertion as specified in [*FIDOSignatureFormat*]. It then returns this assertion to the user agent.

If the authenticator cannot find any credential corresponding to the specified Relying Party that matches the specified criteria, it terminates the operation and returns an error.

If the user refuses consent, the authenticator returns an appropriate error status to the client.

3.3 The *authenticatorCancel* operation

This operation takes no input parameters and returns no result.

When this operation is invoked by the client in an authenticator session, it has the effect of terminating any `authenticatorMakeCredential` or `authenticatorGetAssertion` operation currently in progress in that authenticator session. The authenticator stops prompting for, or accepting, any user input related to authorizing the canceled operation. The client ignores any further responses from the authenticator for the canceled operation.

This operation is ignored if it is invoked in an authenticator session which does not have an `authenticatorMakeCredential` or `authenticatorGetAssertion` operation currently in progress.

4. FIDO Credential API

This section normatively specifies the API for creating and using FIDO 2.0 credentials. Support for deleting credentials is deliberately omitted; this is expected to be done through platform-specific user interfaces rather than from a script. The basic idea is that the credentials belong to the user and are managed by the browser and underlying platform. Scripts can (with the user's consent) request the browser to create a new credential for future use by the script's origin. Scripts can also request the user's permission to perform authentication operations with an existing credential held by the platform. However, all such operations are mediated by the browser and/or platform on the user's behalf. At no point does the script get access to the credentials themselves; it only gets information about the credentials in the form of objects.

User agents *SHOULD* only expose this API to callers in *secure contexts*, as defined in [*powerful-features*].

In the future, this API may be integrated into a more general Web API framework for credential management, which is being worked on in the W3C. Such integration will, most likely, create intermediate interface and dictionary types, from which the types in this specification will then inherit. However the experience of the FIDO developer and end user will not be substantially changed by this. In the meantime, this specification is maintained in a more minimal form for ease of review.

The API is defined by the following Web IDL fragment.

```
partial interface Window {
    readonly    attribute FIDOCredentials fido;
};

interface FIDOCredentials {
    Promise<FIDOCredentialInfo> makeCredential(Account accountInformation,
sequence<FIDOCredentialParameters> cryptoParameters, DOMString
attestationChallenge, optional unsigned long credentialTimeoutSeconds,
optional sequence<Credential> blacklist, optional FIDOExtensions
credentialExtensions);

    Promise<FIDOAssertion>      getAssertion(DOMString assertionChallenge,
optional unsigned long assertionTimeoutSeconds, optional sequence<Credential>
whitelist, optional FIDOExtensions assertionExtensions);
};

interface FIDOCredentialInfo {
    readonly    attribute Credential           credential;
    readonly    attribute AlgorithmIdentifier  algorithm;
    readonly    attribute any                  publicKey;
    readonly    attribute AttestationStatement attestation;
};

dictionary Account {
    required DOMString rpDisplayName;
    required DOMString displayName;
             DOMString name;
             DOMString id;
             DOMString imageUri;
};
```

```
dictionary FIDOCredentialParameters {
    required CredentialType      type;
    required AlgorithmIdentifier algorithm;
};

enum CredentialType {
    "FIDO"
};

interface Credential {
    readonly    attribute CredentialType type;
    readonly    attribute DOMString      id;
};
```

4.1 *FIDOCredentials* Interface

This interface consists of the following methods.

4.1.1 Create a new credential (*makeCredential* method)

With this method, a script can request the user agent to create a new credential of a given type and persist it to the underlying platform, which may involve data storage managed by the browser or the OS. The user agent will prompt the user to approve this operation. On success, the promise will be resolved with a FIDOCredentialInfo object describing the newly created credential.

This method takes the following parameters:

- The *accountInformation* parameter specifies information about the user account for which the credential is being created. This is meant for later use by the authenticator when it needs to prompt the user to select a credential.
- The *cryptoParameters* parameter supplies information about the desired properties of the credential to be created. The sequence is ordered from most preferred to least preferred. The platform makes a best effort to create the most preferred credential that it can.
- The *attestationChallenge* parameter contains a challenge intended to be used for generating the attestation statement of the newly created credential.
- The optional *credentialTimeoutSeconds* parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
- The optional *blacklist* parameter is intended for use by Relying Parties that wish to limit the creation of multiple credentials for the same account on a single authenticator. The platform is requested to return an error if the new credential would be created on an authenticator that also contains one of the credentials enumerated in this parameter.
- The optional *credentialExtensions* parameter contains additional parameters requesting additional processing by the client and authenticator. For example, the caller may request that only authenticators with certain capabilities be used to create the credential, or that additional information be returned in the attestation statement. Alternatively, the caller

may specify an additional message that they would like the authenticator to display to the user. Extensions are defined in [*FIDOSignatureFormat*].

When this method is invoked, the user agent *MUST* execute the following algorithm:

1. If `credentialTimeoutSeconds` was specified, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set *adjustedTimeout* to this adjusted value. If `credentialTimeoutSeconds` was not specified then set *adjustedTimeout* to a platform-specific default.
2. Let *promise* be a new Promise. Return *promise* and start a timer for *adjustedTimeout* seconds. Then asynchronously continue executing the following steps.
3. Set *callerOrigin* to the `origin` of the caller. Derive the RP ID from *callerOrigin* by computing the "public suffix + 1" or "PS+1" (which is also referred to as the "Effective Top-Level Domain plus One" or "eTLD+1") part of *callerOrigin* [*PSL*]. Set *rpId* to the RP ID.
4. Initialize *issuedRequests* to an empty list.
5. Process each element of `cryptoParameters` using the following steps:
    a. Let *current* be the currently selected element of `cryptoParameters`.
    b. If *current.type* does not contain a `CredentialType` supported by this implementation, then stop processing *current* and move on to the next element in `cryptoParameters`.
    c. Let *normalizedAlgorithm* be the result of normalizing an algorithm using the procedure defined in [*WebCryptoAPI*], with *alg* set to *current.algorithm* and *op* set to "generateKey". If an error occurs during this procedure, then stop processing *current* and move on to the next element in `cryptoParameters`.
6. If `blacklist` is undefined, set it to the empty list.
7. If `credentialExtensions` was specified, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. Call this data *clientExtensions*.
8. For each embedded or external authenticator currently available on this platform: asynchronously invoke the `authenticatorMakeCredential` operation on that authenticator with *callerOrigin*, *rpId*, `accountInformation`, *current.type*, *normalizedAlgorithm*, `blacklist`, `attestationChallenge` and *clientExtensions* as parameters. Add a corresponding entry to *issuedRequests*.
9. While *issuedRequests* is not empty, perform the following actions depending upon the *adjustedTimeout* timer and responses from the authenticators:
    a. If the *adjustedTimeout* timer expires, then for each entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator and remove its entry from the list.
    b. If any authenticator returns a status indicating that the user cancelled the operation, delete that authenticator's entry from *issuedRequests*. For each remaining entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator and remove its entry from the list.
    c. If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.

    d. If any authenticator indicates success, create a new `FIDOCredentialInfo` object named *value* and populate its fields with the values returned from the authenticator. Resolve *promise* with *value* and terminate this algorithm.
10. Resolve *promise* with a `DOMException` whose name is "`NotFoundError`", and terminate this algorithm.

During the above process, the user agent *SHOULD* show some UI to the user to guide them in the process of selecting and authorizing an authenticator.

4.1.2 Use an existing credential (`getAssertion` method)

This method is used to discover and use an existing FIDO 2.0 credential, with the user's consent. The script optionally specifies some criteria to indicate what credentials are acceptable to it. The user agent and/or platform locates credentials matching the specified criteria, and guides the user to pick one that the script should be allowed to use. The user may choose not to provide a credential even if one is present, for example to maintain privacy.

This method takes the following parameters:

- The *assertionChallenge* parameter contains a string that the selected authenticator is expected to sign to produce the assertion.
- The optional *assertionTimeoutSeconds* parameter specifies a time, in seconds, that the caller is willing to wait for the call to complete. This is treated as a hint, and may be overridden by the platform.
- The optional *whitelist* member contains a list of credentials acceptable to the caller, in order of the caller's preference.
- The optional *assertionExtensions* parameter contains additional parameters requesting additional processing by the client and authenticator. For example, if transaction confirmation is sought from the user, then the prompt string would be included in an extension. Extensions are defined in a companion specification.

When this method is invoked, the user agent *MUST* execute the following algorithm:

1. If `assertionTimeoutSeconds` was specified, check if its value lies within a reasonable range as defined by the platform and if not, correct it to the closest value lying within that range. Set *adjustedTimeout* to this adjusted value. If `assertionTimeoutSeconds` was not specified then set *adjustedTimeout* to a platform-specific default.
2. Let *promise* be a new Promise. Return *promise* and start a timer for *adjustedTimeout* seconds. Then asynchronously continue executing the following steps.
3. Set *callerOrigin* to the `origin` of the caller. Derive the RP ID from *callerOrigin* by computing the "public suffix + 1" or "PS+1" (which is also referred to as the "Effective Top-Level Domain plus One" or "eTLD+1") part of *callerOrigin* [*PSL*]. Set *rpId* to the RP ID.
4. Initialize *issuedRequests* to an empty list.

5. If `assertionExtensions` was specified, process any extensions supported by this client platform, to produce the extension data that needs to be sent to the authenticator. Call this data *clientExtensions*.
6. For each embedded or external authenticator currently available on this platform, perform the following steps:
    a. If `whitelist` is undefined or empty, let *credentialList* be a list containing a single wildcard entry.
    b. If `whitelist` is defined and non-empty, optionally execute a platform-specific procedure to determine which of these credentials can possibly be present on this authenticator. Set *credentialList* to this filtered list. If *credentialList* is empty, ignore this authenticator and do not perform any of the following per-authenticator steps.
    c. Asynchronously invoke the `authenticatorGetAssertion` operation on this authenticator with *callerOrigin*, *rpId*, `assertionChallenge`, *credentialList*, and *clientExtensions* as parameters.
    d. Add an entry to *issuedRequests*, corresponding to this request.
7. While *issuedRequests* is not empty, perform the following actions depending upon the *adjustedTimeout* timer and responses from the authenticators:
    a. If the timer for *adjustedTimeout* expires, then for each entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator and remove its entry from the list.
    b. If any authenticator returns a status indicating that the user cancelled the operation, delete that authenticator's entry from *issuedRequests*. For each remaining entry in *issuedRequests* invoke the `authenticatorCancel` operation on that authenticator, and remove its entry from the list.
    c. If any authenticator returns an error status, delete the corresponding entry from *issuedRequests*.
    d. If any authenticator returns success, create a new `FIDOAssertion` object named *value* and populate its fields with the values returned from the authenticator. Resolve *promise* with `value` and terminate this algorithm.
8. Resolve *promise* with a `DOMException` whose name is "`NotFoundError`", and terminate this algorithm.

During the above process, the user agent *SHOULD* show some UI to the user to guide them in the process of selecting and authorizing an authenticator with which to complete the operation.

4.2 *`FIDOCredentialInfo`* Interface

This interface represents a newly-created FIDO credential. It contains information about the credential that can be used to locate it later for use, and also contains metadata that can be used by the FIDO Relying Party to assess the strength of the credential during registration.

The *`credential`* attribute contains a unique identifier for the credential represented by this object.

The `algorithm` attribute contains the cryptographic algorithm associated with the credential, in the format defined in [*WebCryptoAPI*].

The `publicKey` attribute contains the public key associated with the credential, represented as a JsonWebKey structure as defined in [*WebCryptoAPI*].

The `attestation` attribute contains a key attestation statement returned by the authenticator. This provides information about the credential and the authenticator it is held in, such as the level of security assurance provided by the authenticator.

4.3 User Account Information (dictionary `Account`)

This dictionary is used by the caller to specify information about the user account and Relying Party with which a credential is to be associated. It is intended to help the authenticator in providing a friendly credential selection interface for the user.

The `rpDisplayName` member contains the friendly name of the Relying Party, such as "Google", "Microsoft" or "PayPal".

The `displayName` member contains the friendly name associated with the user account by the Relying Party, such as "John P. Smith".

The `name` member contains a detailed name for the account, such as "john.p.smith@example.com".

The `id` member contains an identifier for the account, stored for the use of the Relying Party. This is not meant to be displayed to the user.

The `imageUri` member contains a URI that resolves to the user's account image. This may be a URL that can be used to retrieve the user's current avatar, or a data URI that contains the image data.

4.4 Parameters for Credential Generation (dictionary `FIDOCredentialParameters`)

This dictionary is used to supply additional parameters when creating a new credential.

The `type` member specifies the type of credential to be created.

The `algorithm` member specifies the cryptographic algorithm with which the newly generated credential will be used.

4.5 Supporting Data Structures

The FIDO credential type uses certain data structures that are specified in supporting specifications. These are as follows.

4.5.1 Credential Type enumeration (enum *CredentialType*)

This enumeration defines the valid credential types. It is an extension point; values may be added to it in the future, as more credential types are defined. The values of this enumeration are used for versioning the FIDO assertion and attestation statement according to the type of the authenticator.

Currently one credential type is defined, namely "*FIDO*", the FIDO 2.0 credential type.

4.5.2 Unique Identifier for Credential (interface *Credential*)

This interface contains the attributes that are returned to the caller when a new credential is created, and can be used later by the caller to select a credential for use.

The `type` attribute indicates the specification and version that this credential conforms to.

The `id` attribute contains an identifier for the credential, chosen by the platform with help from the authenticator. This identifier is used to look up credentials for use, and is therefore expected to be globally unique with high probability across all credentials of the same type. This API does not constrain the format or length of this identifier, except that it must be sufficient for the platform to uniquely select a key. For example, an authenticator without on-board storage may create identifiers that consist of the key material wrapped with a key that is burned into the authenticator.

4.5.3 Cryptographic Algorithm Identifier (type AlgorithmIdentifier)

A string or dictionary identifying a cryptographic algorithm and optionally a set of parameters for that algorithm. This type is defined in [*WebCryptoAPI*].

4.5.4 FIDO Assertion (interface *FIDOAssertion*)

FIDO 2.0 credentials produce a cryptographic signature that provides proof of possession of a private key as well as evidence of user consent to a specific transaction. The structure of these signatures is defined in [*FIDOSignatureFormat*].

4.5.5 FIDO Assertion Extensions (dictionary *FIDOExtensions*)

This is a dictionary containing zero or more extensions as defined in [*FIDOSignatureFormat*]. An extension is an additional parameter that can be passed to the getAssertion method and triggers some additional processing by the client platform and/or the authenticator.

If the caller wants to pass extensions to the platform, it *SHOULD* do so by adding one entry per extension to this `FIDOExtensions` dictionary with the extension identifier as the key, and the extension's value as the value (see [*FIDOSignatureFormat*] for details).

4.5.6 Key Attestation Statement (interface *AttestationStatement*)

FIDO 2.0 authenticators also provide some form of key attestation. The basic requirement is that the authenticator can produce, for each credential public key, attestation information that can be verified by a Relying Party. Typically this information contains a signature by an attesting key over the attested public key and a challenge, as well as a certificate or similar information providing provenance information for the attesting key, enabling a trust decision to be made. The structure of these attestation statements is defined in [*FIDOKeyAttestation*].

5. Sample scenarios

This section is non-normative.

In this section, we walk through some events in the lifecycle of a FIDO 2.0 credential, along with the corresponding sample code for using this API. Note that this is an example flow, and does not limit the scope of how the API can be used.

As was the case in earlier sections, this flow focuses on a use case involving an external first-factor authenticator with its own display. One example of such an authenticator would be a smart phone. Other authenticator types are also supported by this API, subject to implementation by the platform. For instance, this flow also works without modification for the case of an authenticator that is embedded in the client platform. The flow also works for the case of an external authenticator without its own display (similar to a smart card) subject to specific implementation considerations. Specifically, the client platform needs to display any prompts that would otherwise be shown by the authenticator, and the authenticator needs to allow the client platform to enumerate all the authenticator's credentials so that the client can have information to show appropriate prompts.

5.1 Registration

This is the first time flow, when a new credential is created and registered with the server.

1. The user visits example.com, which serves up a script. At this point, the user must already be logged in using a legacy username and password, or additional authenticator, or other means acceptable to the Relying Party.
2. The Relying Party script runs the code snippet below.
3. The client platform searches for and locates the external authenticator.
4. The client platform connects to the external authenticator, performing any pairing actions if necessary.
5. The external authenticator shows appropriate UI for the user to select the authenticator on which the new credential will be created, and obtains a biometric or other authorization gesture from the user.
6. The external authenticator returns a response to the client platform, which in turn returns a response to the RP script. If the user declined to select an authenticator or provide authorization, an appropriate error is returned.
7. If a new credential was created,

a. The RP script sends the newly generated public key to the server, along with additional information about public key such as attestation that it is held in trusted hardware.
b. The server stores the public key in its database and associates it with the user as well as with the strength of authentication indicated by attestation, also storing a friendly name for later use.
c. The script may store data such as the credential ID in local storage, to improve future UX by narrowing the choice of credential for the user.

The sample code for generating and registering a new key follows:

Example 1

```
var fidoAPI = window.fido;

if (!fidoAPI) { /* Platform not capable. Handle error. */ }

var userAccountInformation = {
  rpDisplayName: "PayPal",
  displayName: "John P. Smith",
  name: "johnpsmith@gmail.com",
  id: "1098237235409872";
  imageUri: "https://pics.paypal.com/00/p/aBjjjpqPb.png"
};

// This RP will accept either an ES256 or RS256 credential, but
// prefers an ES256 credential.
var cryptoParams = [
  {
    type: "FIDO",
    algorithm: "ES256",
  },
  {
    type: "FIDO",
    algorithm: "RS256",
  }
];
var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var timeoutSeconds = 300;  // 5 minutes
var blacklist = [];  // No blacklist
var extensions = {"fido.location": true};  // Include location information
                                           // in attestation


// Note: The following call will cause the authenticator to display UI.
fidoAPI.makeCredential(userAccountInformation, cryptoParams, challenge,
                       timeoutSeconds, blacklist, extensions)
  .then(function (newCredentialInfo) {
    // Send new credential info to server for verification and registration.
}).catch(function (err) {
    // No acceptable authenticator or user refused consent. Handle
appropriately.
});
```

5.2 Authentication

This is the flow when a user with an already registered credential visits a website and wants to authenticate using the credential.

1. The user visits example.com, which serves up a script.
2. The script asks the client platform for a FIDO identity assertion, providing as much information as possible to narrow the choice of acceptable credentials for the user. This may be obtained from the data that was stored locally after registration, or by other means such as prompting the user for a username.
3. The Relying Party script runs one of the code snippets below.
4. The client platform searches for and locates the external authenticator.
5. The client platform connects to the external authenticator, performing any pairing actions if necessary.
6. The external authenticator presents the user with a notification that their attention is required. On opening the notification, the user is shown a friendly selection menu of acceptable credentials using the account information provided when creating the credentials, along with some information on the origin that is requesting these keys.
7. The authenticator obtains a biometric or other authorization gesture from the user.
8. The external authenticator returns a response to the client platform, which in turn returns a response to the RP script. If the user declined to select a credential or provide an authorization, an appropriate error is returned.
9. If an assertion was successfully generated and returned,
    a. The script sends the assertion to the server.
    b. The server examines the assertion and validates that it was correctly generated. If so, it looks up the identity associated with the associated public key; that identity is now authenticated. If the public key is not recognized by the server (e.g., deregistered by server due to inactivity) then the authentication has failed; each Relying Party will handle this in its own way.
    c. The server now does whatever it would otherwise do upon successful authentication — return a success page, set authentication cookies, etc.

If the Relying Party script does not have any hints available (e.g., from locally stored data) to help it narrow the list of credentials, then the sample code for performing such an authentication might look like this:

Example 2

```
var fidoAPI = window.fido;

if (!fidoAPI) { /* Platform not capable. Handle error. */ }

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var timeoutSeconds = 300;  // 5 minutes
var whitelist = [{ type: "FIDO" }];

fidoAPI.getAssertion(challenge, timeoutSeconds, whitelist)
  .then(function (assertion) {
    // Send assertion to server for verification
```

```
}).catch(function (err) {
    // No acceptable credential or user refused consent. Handle
appropriately.
});
```

On the other hand, if the Relying Party script has some hints to help it narrow the list of credentials, then the sample code for performing such an authentication might look like the following. Note that this sample also demonstrates how to use the extension for transaction authorization.

Example 3

```
var fidoAPI = window.fido;

if (!fidoAPI) { /* Platform not capable. Handle error. */ }

var challenge = "Y2xpbWIgYSBtb3VudGFpbg";
var timeoutSeconds = 300;  // 5 minutes
var acceptableCredential1 = {
  type: "FIDO",
  id: "ISEhISEhIWhpIHRoZXJlISEhISEhIQo=",
};
var acceptableCredential2 = {
  type: "FIDO",
  id: "cm9zZXMgYXJlIHJlZCwgdmlvbGV0cyBhcmUgYmx1ZQo=",
};
var whitelist = [acceptableCredential1, acceptableCredential2];
var extensions = { 'fido.txauth.simple':
                   "Wave your hands in the air like you just don't care" };

fidoAPI.getAssertion(challenge, timeoutSeconds, whitelist, extensions)
  .then(function (assertion) {
    // Send assertion to server for verification
}).catch(function (err) {
    // No acceptable credential or user refused consent. Handle
appropriately.
});
```

5.3 Decommissioning

The following are possible situations in which decommissioning a credential might be desired. Note that all of these are handled on the server side and do not need support from the API specified here.

- Possibility #1 — user reports the credential as lost.
    - User goes to server.example.net, authenticates and follows a link to report a lost/stolen device.
    - Server returns a page showing the list of registered credentials with friendly names as configured during registration.
    - User selects a credential and the server deletes it from its database.
    - In future, Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.

- Possibility #2 — server deregisters the credential due to inactivity.
    - Server deletes credential from its database during maintenance activity.
    - In the future, the Relying Party script does not specify this credential in any list of acceptable credentials, and assertions signed by this credential are rejected.
- Possibility #3 — user deletes the credential from the device.
    - User employs a device-specific method (e.g., device settings UI) to delete a credential from their device.
    - From this point on, this credential will not appear in any selection prompts, and no assertions can be generated with it.
    - Sometime later, the server deregisters this credential due to inactivity.

6. Acknowledgements

This section is non-normative.

We would like to thank the following for their contributions to, and thorough review of, this specification: Jing Jin, Michael B. Jones, Rolf Lindemann.

A. References

A.1 Normative references

[DOM4]
Anne van Kesteren; Aryeh Gregor; Ms2ger; Alex Russell; Robin Berjon. *W3C DOM4*. 6 October 2015. W3C Proposed Recommendation. URL: http://www.w3.org/TR/dom/

[FIDOKeyAttestation]
FIDO 2.0: Key attestation format. URL: http://www.w3.org/Submission/2015/SUBM-fido-key-attestation-20151120/

[FIDOSignatureFormat]
FIDO 2.0: Signature format. URL: http://www.w3.org/Submission/2015/SUBM-fido-signature-format-20151120/

[HTML5]
Ian Hickson; Robin Berjon; Steve Faulkner; Travis Leithead; Erika Doyle Navara; Edward O'Connor; Silvia Pfeiffer. *HTML5*. 28 October 2014. W3C Recommendation. URL: http://www.w3.org/TR/html5/

[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

[WebCryptoAPI]
Ryan Sleevi; Mark Watson. *Web Cryptography API*. 11 December 2014. W3C Candidate Recommendation. URL: http://www.w3.org/TR/WebCryptoAPI/

[WebIDL-1]
Cameron McCormack; Boris Zbarsky. *WebIDL Level 1*. 4 August 2015. W3C Working Draft. URL: http://www.w3.org/TR/WebIDL-1/

A.2 Informative references

[PSL]
Public Suffix List. Mozilla Foundation.

[powerful-features]
Mike West; Yan Zhu. *Privileged Contexts*. 24 April 2015. W3C Working Draft. URL: http://www.w3.org/TR/powerful-features/