W3C Working Draft

Credential Management Level 1

W3C Working Draft, 17 January 2019

This version:
https://www.w3.org/TR/2019/WD-credential-management-1-20190117/

Latest published version:
https://www.w3.org/TR/credential-management-1/

Editor's Draft:
https://w3c.github.io/webappsec-credential-management/

Previous Versions:
https://www.w3.org/TR/2017/WD-credential-management-1-20170804/

Version History:
https://github.com/w3c/webappsec-credential-management/commits/master/index.src.html

Feedback:
public-webappsec@w3.org with subject line “`[credential-management]` *… message topic …*” (archives)

Editor:
Mike West (Google Inc.)

Participate:
File an issue (open issues)

Abstract

This specification describes an imperative API enabling a website to request a user’s credentials from a user agent, and to help the user agent correctly store user credentials for future use.

Status of this document

Table of Contents

§ 1. Introduction

This section is not normative.

Signing into websites is more difficult than it should be. The user agent is in a unique position to improve the experience in a number of ways, and most modern user agents have recognized this by providing some measure of credential management natively in the browser. Users can save usernames and passwords for websites, and those credentials are autofilled into sign-in forms with varying degrees of success.

The `autocomplete` attribute offers a declarative mechanism by which websites can work with user agents to improve the latter's ability to detect and fill sign-in forms by marking specific fields as "username" or "password", and user agents implement a wide variety of detection heuristics to work with websites which haven't taken the time to provide this detail in markup.

While this combination of heuristic and declarative detection works relatively well, the status quo leaves some large gaps where detection is problematic. Sites with uncommon sign-in mechanisms (submitting credentials via `XMLHttpRequest` [XMLHTTPREQUEST], for instance) are difficult to reliably detect, as is the increasingly common case in which users wish to authenticate themselves using a federated identity provider. Allowing websites to more directly interact with the user agent's credential manager would allow the credential manager to be more accurate on the one hand, and to assist users with federated sign-in on the other.

These use cases are explored in more detail in §1.1 Use Cases and in Credential Management: Use Cases and Requirements; this specification attempts to address many of the requirements that document outlines by defining a Credential Manager API which a website can use to request credentials for a user, and to ask the user agent to persist credentials when a user signs in successfully.

Note: The API defined here is intentionally small and simple: it does not intend to provide authentication in and of itself, but is limited to providing an interface to the existing credential managers implemented by existing user agents. That functionality is valuable *right now*, without significant effort on the part of either vendors or authors. There's certainly quite a bit more which could be done, of course. See §8 Future Work for some thoughts we've punted for now, but which could be explored in future iterations of this API.

§ 1.1. Use Cases

Modern user agents generally offer users the capability to save passwords when signing into a website, and likewise offer the capability to fill those passwords into sign-in forms fully- or semi-automatically when users return to a website. From the perspective of a website, this behavior is completely invisible: the website doesn't know that passwords have been stored, and it isn't notified that passwords have been filled. This is both good and bad. On the one hand, a user agent's password manager works regardless of whether or not a site cooperates, which is excellent for users. On the other, the password managers' behaviors are a fragile and proprietary hodgepodge of heuristics meant to detect and fill sign-in forms, password change forms, etc.

A few problems with the status quo stand out as being particularly noteworthy:

- User agents have an incredibly difficult time helping users with federated identity providers. While detecting a username/password form submission is fairly straightforward, detecting sign-in via a third-party is quite difficult to reliably do well. It would be nice if a website could help the user agent understand the intent behind the redirects associated with a typical federated sign-in action.
- Likewise, user agents struggle to detect more esoteric sign-in mechanisms than simple username/password forms. Authors increasingly asynchronously sign users in via `XMLHttpRequest` or similar mechanisms in order to improve the experience and take more control over the presentation. This is good for users, but tough for user agents to integrate into their password managers. It would be nice if a website could help the user agent make sense of the sign-in mechanism they choose to use.

- Finally, changing passwords is less well-supported than it could be if the website explicitly informed the user agent that credentials had changed.

§ 2. Core API

From a developer’s perspective, a ***credential*** is an object which allows a developer to make an authentication decision for a particular action. This section defines a generic and extensible `Credential` interface which serves as a base class for credentials defined in this and other documents, along with a set of APIs hanging off of `navigator.credentials.*` which enable developers to obtain them.

Various types of credentials are represented to JavaScript as an interface which inherits, either directly or indirectly, from the `Credential` interface. This document defines two such interfaces, `PasswordCredential` and `FederatedCredential`.

A credential is ***effective*** for a particular origin if it is accepted as authentication on that origin. Even if a credential is effective at a particular point in time, the UA can’t assume that the same credential will be effective at any future time, for a couple reasons:

1. A password credential may stop being effective if the account holder changes their password.
2. A credential made from a token received over SMS is likely to only be effective for a single use.

Single-use credentials are generated by a ***credential source***, which could be a private key, access to a federated account, the ability to receive SMS messages at a particular phone number, or something else. Credential sources are not exposed to Javascript or explicitly represented in this specification. To unify the model, we consider a password to be a credential source on its own, which is simply copied to create password credentials.

Even though the UA can’t assume that an effective credential will still be effective if used a second time, or that a credential source that has generated an effective credential will be able to generate a second effective credential in the future, the second is more likely than the first. By recording (with `store()`) which credentials have been effective in the past, the UA has a better chance of offering effective credential sources to the user in the future.

§ 2.1. Infrastructure

User agents MUST internally provide a ***credential store***, which is a vendor-specific, opaque storage mechanism to record which credentials have been effective. It offers the following capabilities for credential access and persistence:

1. ***Store a credential*** for later retrieval. This accepts a credential, and inserts it into the credential store.
2. ***Retrieve a list of credentials***. This accepts an arbitrary filter, and returns a set of credentials that match the filter.
3. ***Modify a credential***. This accepts a credential, and overwrites the state of an existing credential in the credential store.

Additionally, the credential store should maintain a ***`prevent silent access` flag*** for origins (which is set to `true` unless otherwise specified). An origin ***requires user mediation*** if its flag is set to `true`.

Note: The importance of user mediation is discussed in more detail in §5 User Mediation.

Note: The credential store is an internal implementation detail of a user agent's implementation of the API specified in this document, and is not exposed to the web directly. More capabilities may be specified by other documents in support of specific credential types.

This document depends on the Infra Standard for a number of foundational concepts used in its algorithms and prose [INFRA].

An environment settings object (*settings*) is ***same-origin with its ancestors*** if the following algorithm returns `true`:

1. If *settings* has no responsible browsing context, return `false`.
2. Let *origin* be *settings*' origin.

3. Let *current* be *settings*’ responsible browsing context.
4. While *current* has a parent browsing context:
    1. Set *current* to *current*’s parent browsing context.
    2. If *current*’s active document's origin is not same origin with *origin*, return `false`.
5. Return `true`.

§ 2.2. The `Credential` Interface

```
[Exposed=Window, SecureContext]
interface Credential {
  readonly attribute USVString id;
  readonly attribute DOMString type;
};
```

***`id`*, of type USVString, readonly**

The credential’s identifier. The requirements for the identifier are distinct for each type of credential. It might represent a username for username/password tuples, for example.

***`type`*, of type DOMString, readonly**

This attribute’s getter returns the value of the object’s interface object's `[[type]]` slot, which specifies the credential type represented by this object.

`[[type]]`

The `Credential` interface object has an internal slot named `[[type]]`, which unsurprisingly contains a string representing the ***credential type***. The slot’s value is the empty string unless otherwise specified.

Note: The `[[type]]` slot's value will be the same for all credentials implementing a particular interface, which means that developers can rely on `obj.type` returning a string that unambiguously represents the specific kind of `Credential` they're dealing with.

`[[discovery]]`

The `Credential` interface object has an internal slot named `[[discovery]]`, representing the mechanism by which the user agent can collect credentials of a given type. Its value is either "***`credential store`***" or "***`remote`***". The former value means that all available credential information is stored in the user agent's credential store, while the latter means that the user agent can discover credentials outside of those explicitly represented in the credential store via interaction with some external device or service.

ISSUE 1 Talk to Tobie/Dominic about the interface object bits, here and in §2.5.1 Request a Credential, etc. I'm not sure I've gotten the terminology right. interface prototype object, maybe?

Some `Credential` objects are ***origin bound***: these contain an internal slot named ***`[[origin]]`***, which stores the origin for which the `Credential` may be effective.

2.2.1. `Credential` Internal Methods

The `Credential` interface object features several internal methods facilitating retrieval and storage of `Credential` objects, with default "no-op" implementations as specified in this section, below.

Unless otherwise specified, each interface object created for interfaces which inherit from `Credential` MUST provide implementations for at least one of these internal methods, overriding `Credential`'s default implementations, as appropriate for the credential type. E.g., §3.2 The PasswordCredential Interface, §4.1 The FederatedCredential Interface, and [WEBAUTHN].

2.2.1.1. `[[CollectFromCredentialStore]]` internal method

`[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` is called with an origin, a `CredentialRequestOptions`, and a boolean which is true iff the caller’s environment settings object is same-origin with its ancestors. The algorithm returns a set of `Credential` objects from the user agent’s credential store that match the options provided. If no matching `Credential` objects are available, the returned set will be empty.

`Credential`'s default implementation of `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`:

1. Return an empty set.

2.2.1.2. `[[DiscoverFromExternalSource]]` internal method

`[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` is called in parallel with an origin, a `CredentialRequestOptions` object, and a boolean which is true iff the caller’s environment settings object is same-origin with its ancestors. It returns a `Credential` if one can be returned given the options provided, `null` if no credential is available, or an error if discovery fails (for example, incorrect options could produce a `TypeError`). If this kind of `Credential` is only effective for a single use or a limited time, this method is responsible for generating new credentials using a credential source.

`Credential`'s default implementation of `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`:

1. Return `null`.

2.2.1.3. `[[Store]]` internal method

`[[Store]](credential, sameOriginWithAncestors)` is called in parallel with a `Credential`, and a boolean which is true iff the caller’s environment settings object is same-origin with its ancestors. The algorithm returns once `Credential` is persisted to the credential store.

`Credential`'s default implementation of `[[Store]](credential, sameOriginWithAncestors)`:

1. Return `undefined`.

2.2.1.4. `[[Create]]` internal method

`[[Create]](origin, options, sameOriginWithAncestors)` is called in parallel with an origin, a `CredentialCreationOptions`, and a boolean which is true iff the caller’s environment settings object is same-origin with its ancestors. The algorithm either:

- creates a `Credential`, or
- does not create a credential and returns `null`, or
- returns an error if creation fails due to exceptional situations (for example, incorrect options could produce a `TypeError`).

When creating a `Credential`, it will return an algorithm that takes a global object and returns an interface object inheriting from `Credential`. This algorithm MUST be invoked from a task.

Note: This algorithm’s steps are defined on a per-credential type basis.

`Credential`'s default implementation of `[[Create]](origin, options, sameOriginWithAncestors)`:

1. Return `null`.

§ 2.2.2. `CredentialUserData` Mixin

Some `Credential` objects contain data which aims to give users a human-readable disambiguation mechanism in the credential chooser by providing a friendly name and icon:

```
[SecureContext]
interface mixin CredentialUserData {
  readonly attribute USVString name;
  readonly attribute USVString iconURL;
};
```

***`name`*, of type USVString, readonly**
A name associated with the credential, intended as a human-understandable public name for display in a credential chooser.

***`iconURL`*, of type USVString, readonly**
A URL pointing to an image for the credential, intended for display in a credential chooser. This URL MUST be an a priori authenticated URL.

§ 2.3. `navigator.credentials`

Developers retrieve `Credential`s and interact with the user agent’s credential store via methods exposed on the `CredentialsContainer` interface, which hangs off the `Navigator` object as `navigator.credentials`.

```
partial interface Navigator {
  [SecureContext, SameObject] readonly attribute CredentialsContainer credentials;
};
```

The ***credentials*** attribute MUST return the `CredentialsContainer` associated with the context object.

Note: As discussed in §6.3 Insecure Sites, the credential management API is exposed only in Secure Contexts.

```
[Exposed=Window, SecureContext]
interface CredentialsContainer {
  Promise<Credential?> get(optional CredentialRequestOptions options);
  Promise<Credential> store(Credential credential);
  Promise<Credential?> create(optional CredentialCreationOptions options);
  Promise<void> preventSilentAccess();
};

dictionary CredentialData {
  required USVString id;
};
```

get(options)

When `get()` is called, the user agent MUST return the result of executing Request a `Credential` on `options`.

Arguments for the CredentialsContainer.get(options) method.

Parameter	Type	Nullable	Optional	Description
options	CredentialRequestOptions	✘	✔	The set of properties governing the scope of the request.

store(credential)

When `store()` is called, the user agent MUST return the result of executing Store a `Credential` on `credential`.

Arguments for the CredentialsContainer.store(credential) method.

Parameter	Type	Nullable	Optional	Description
credential	Credential	✘	✘	The credential to be stored.

create(options)

When `create()` is called, the user agent MUST return the result of executing Create a `Credential` on `options`.

Arguments for the CredentialsContainer.create(options) method.

Parameter	Type	Nullable	Optional	Description
options	CredentialCreationOptions	✘	✔	The options used to create a `Credential`.

preventSilentAccess()

When `preventSilentAccess()` is called, the user agent MUST return the result of executing Prevent Silent Access on the current settings object.

Note: The intent here is a signal from the origin that the user has signed out. That is, after a click on a "Sign out" button, the site updates the user's session info, and calls `navigator.credentials.preventSilentAccess()`. This sets the `prevent silent access` flag, meaning that credentials will not be automagically handed back to the page next time the user visits.

Note: This function was previously called `requireUserMediation()` which should be considered deprecated.

When a `Navigator` object (*navigator*) is created, the user agent MUST create a new `CredentialsContainer` object, using *navigator*'s relevant Realm, and associate it with *navigator*.

§ 2.3.1. The `CredentialRequestOptions` Dictionary

In order to retrieve a `Credential` via `get()`, the caller specifies a few parameters in a `CredentialRequestOptions` object.

Note: The `CredentialRequestOptions` dictionary is an extension point. If and when new types of credentials are introduced that require options, their dictionary types will be added to the dictionary so they can be passed into the request. See §7.2 Extension Points.

```
dictionary CredentialRequestOptions {
  CredentialMediationRequirement mediation = "optional";
  AbortSignal signal;
};
```

***`mediation`*, of type CredentialMediationRequirement, defaulting to `"optional"`**

This property specifies the mediation requirements for a given credential request. The meaning of each enum value is described below in `CredentialMediationRequirement`. Processing details are defined in §2.5.1 Request a Credential.

***`signal`*, of type AbortSignal**

This property lets the developer abort an ongoing `get()` operation. An aborted operation may complete normally (generally if the abort was received after the operation finished) or reject with an "`AbortError`" `DOMException`."

Earlier versions of this specification defined a boolean `unmediated` member. Setting that to `true` had the effect of setting `mediation` to "`silent`", and setting it to `false` had the effect of setting `mediation` to "`optional`".

`unmediated` should be considered deprecated; new code should instead rely on `mediation`.

The ***relevant credential interface objects*** for a given `CredentialRequestOptions` (*options*) is a set of interface objects, collected as follows:

1. Let *settings* be the current settings object
2. Let *interface objects* be the set of interface objects on *settings*' global object.
3. Let *relevant interface objects* be an empty set.
4. For each *object* in *interface objects*:
    1. If *object*'s inherited interfaces do not contain `Credential`, continue.
    2. Let *key* be *object*'s `[[type]]` slot's value.
    3. If *options*[*key*] exists, append *object* to *relevant interface objects*.

ISSUE 2 jyasskin@ suggests replacing the iteration through the interface objects with a registry. I'm not sure which is less clear, honestly. I'll leave it like this for the moment, and we can argue about whether this is too much of a `COMEFROM` interface.

A given `CredentialRequestOptions` *options* is ***matchable a priori*** if the following steps return `true`:

1. For each *interface* in *options*' relevant credential interface objects:
    1. If *interface*'s `[[discovery]]` slot's value is not "`credential store`", return `false`.
2. Return `true`.

Note: When executing `get(options)`, we only return credentials without user mediation if the provided `CredentialRequestOptions` is matchable a priori. If any credential types are requested that could require discovery from some external service (OAuth tokens, security key authenticators, etc.), then user mediation will be required in order to guide the discovery process (by choosing a federated identity provider, BTLE device, etc).

§ 2.3.2. Mediation Requirements

When making a request via `get(options)`, developers can set a case-by-case requirement for user mediation by choosing the appropriate `CredentialMediationRequirement` enum value.

Note: The §5 User Mediation section gives more detail on the concept in general, and its implications on how the user agent deals with individual requests for a given origin).

```
enum CredentialMediationRequirement {
  "silent",
  "optional",
  "required"
};
```

`silent`

User mediation is suppressed for the given operation. If the operation can be performed without user involvement, wonderful. If user involvement is necessary, then the operation will return `null` rather than involving the user.

> Note: The intended usage is to support "Keep me signed-into this site" scenarios, where a developer may wish to silently obtain credentials if a user should be automatically signed in, but to delay bothering the user with a sign-in prompt until they actively choose to sign-in.

`optional`

If credentials can be handed over for a given operation without user mediation, they will be. If user mediation is required, then the user agent will involve the user in the decision.

> Note: This is the default behavior for `get()`, and is intended to serve a case where a developer has reasonable confidence that a user expects to start a sign-in operation. If a user has just clicked "sign-in" for example, then they won't be surprised or confused to see a credential chooser if necessary.

`required`

The user agent will not hand over credentials without user mediation, even if the prevent silent access flag is unset for an origin.

> Note: This requirement is intended to support reauthentication or user-switching scenarios. Further, the requirement is tied to a specific operation, and does not affect the prevent silent access flag for the origin. To set that flag, developers should call `preventSilentAccess()`.

§ 2.3.2.1. *Examples*

EXAMPLE 1

MegaCorp, Inc. wishes to seamlessly sign in users when possible. They can do so by calling `get()` for all non-signed in users at some convinient point while a landing page is loading, passing in a `mediation` member set to "`silent`". This ensures that users who have opted-into dropping the requirements for user mediation (as described in §5.2 Requiring User Mediation) are signed in, and users who haven't opted-into such behavior won't be bothered with a confusing credential chooser popping up without context:

```
window.addEventListener('load', async () => {
  const credentials = await navigator.credentials.get({
    ...,
    mediation: 'silent'
  });
  if (credentials) {
    // Hooray! Let's sign the user in using these credentials!
  }
});
```

EXAMPLE 2

When a user clicks "Sign In", MegaCorp, Inc. wishes to give them the smoothest possible experience. If they have opted-into signing in without user mediation, and the user agent can unambigiously choose a credential, great! If not, a credential chooser will be presented.

```
document.querySelector('#sign-in').addEventListener('click', async () => {
  const credentials = await navigator.credentials.get({
    ...,
    mediation: 'optional'
  });
  if (credentials) {
    // Hooray! Let’s sign the user in using these credentials!
  }
});
```

Note: MegaCorp, Inc. could also have left off the `mediation` member entirely, as "`optional`" is its default.

EXAMPLE 3

MegaCorp, Inc. wishes to protect a sensitive operation by requiring a user to reauthenticate before taking action. Even if a user has opted-into signing in without user mediation, MegaCorp, Inc. can ensure that the user agent requires mediation by calling `get()` with a `mediation` member set to "`required`":

Note: Depending on the security model of the browser or the credential type, this may require the user to authenticate themselves in some way, perhaps by entering a master password, scanning a fingerprint, etc. before a credential is handed to the website.

```
document.querySelector('#important-form').addEventListener('submit', async () => {
  const credentials = await navigator.credentials.get({
    ...,
    mediation: 'required'
  });
  if (credentials) {
    // Verify that |credentials| enables access, and cancel the submission
    // if it doesn’t.
  } else {
    e.preventDefault();
  }
});
```

EXAMPLE 4

MegaCorp, Inc. wishes to support signing into multiple user accounts at once. In order to ensure that the user gets a chance to select a different credential, MegaCorp, Inc. can call `get()` with a `mediation` member set to "`required`" in order to ensure that that credentials aren't returned automatically in response to clicking on an "Add account" button:

```
document.querySelector('#switch-button').addEventListener('click', e => {
  var c = await navigator.credentials.get({
    ...,
    mediation: 'required'
  });
  if (c) {
    // Sign the user in using |c|.
  }
});
```

§ 2.4. The `CredentialCreationOptions` Dictionary

In order to create a `Credential` via `create()`, the caller specifies a few parameters in a `CredentialCreationOptions` object.

Note: The `CredentialCreationOptions` dictionary is an extension point. If and when new types of credentials are introduced, they will add to the dictionary so they can be passed into the creation method. See §7.2 Extension Points, and the extensions introduced in this document: §3.2 The PasswordCredential Interface and §4.1 The FederatedCredential Interface.

```
dictionary CredentialCreationOptions {
  AbortSignal signal;
};
```

`signal`, of type AbortSignal

This property lets the developer abort an ongoing `create()` operation. An aborted operation may complete normally (generally if the abort was received after the operation finished) or reject with an "`AbortError`" `DOMException`."

§ 2.5. Algorithms

§ 2.5.1. Request a `Credential`

The ***Request a `Credential`*** algorithm accepts a `CredentialRequestOptions` (*options*), and returns a `Promise` that resolves with a `Credential` if one can be unambigiously obtained, or with `null` if not.

1. Let *settings* be the current settings object
2. Assert: *settings* is a secure context.
3. If `options.signal`'s aborted flag is set, then return a promise rejected with an "`AbortError`" `DOMException`.
4. Let *p* be a new promise.
5. Let *origin* be the current settings object's origin.
6. Let *sameOriginWithAncestors* be `true` if *settings* is same-origin with its ancestors, and `false` otherwise.
7. Run the following steps in parallel:
    1. Let *credentials* be the result of collecting `Credential`s from the credential store, given *origin*, *options*, and *sameOriginWithAncestors*.

2. If *credentials* is an exception, reject *p* with *credentials*.
3. If all of the following statements are true, resolve *p* with *credentials*[0] and skip the remaining steps:
    1. *credentials*' size is 1
    2. *origin* does not require user mediation
    3. *options* is matchable a priori.
    4. *options*.`mediation` is not "`required`".

    ISSUE 3 This might be the wrong model. It would be nice to support a site that wished to accept either username/passwords or webauthn-style credentials without forcing a chooser for those users who use the former, and who wish to remain signed in.

4. If *options*' `mediation` is "`silent`", resolve *p* with `null`, and skip the remaining steps.
5. Let *choice* be the result of asking the user to choose a `Credential`, given *options* and *credentials*.
6. If *choice* is `null` or a `Credential`, resolve *p* with *choice* and skip the remaining steps.
7. Assert: *choice* is an interface object.
8. Let *result* be the result of executing *choice*'s `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, given *origin*, *options*, and *sameOriginWithAncestors*.
9. If *result* is a `Credential` or `null`, resolve *p* with *result*.

    Otherwise, reject *p* with *result*.

8. Return *p*.

§ 2.5.2. Collect `Credentials` from the credential store

Given an origin (*origin*), a `CredentialRequestOptions` (*options*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*), the user agent may ***collect `Credential`s from the credential store***, returning a set of `Credential` objects stored by the user agent locally that match *options*' filter. If no such `Credential` objects are known, the returned set will be empty:

1. Let *possible matches* be an empty set.
2. For each *interface* in *options*' relevant credential interface objects:
    1. Let *r* be the result of executing *interface*'s `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` internal method on *origin*, *options*, and *sameOriginWithAncestors*.
    2. If *r* is an exception, return *r*.
    3. Assert: *r* is a list of interface objects.
    4. For each *c* in *r*:
        1. Append *c* to *possible matches*.
3. Return *possible matches*.

2.5.3. Store a `Credential`

The ***Store a `Credential`*** algorithm accepts a `Credential` (*credential*), and returns a `Promise` which resolves once the object is persisted to the credential store.

1. Let *settings* be the current settings object
2. Assert: *settings* is a secure context.
3. Let *sameOriginWithAncestors* be `true` if the current settings object is same-origin with its ancestors, and `false` otherwise.

4. Let *p* be a new promise.
5. Run the following steps in parallel:
    1. Let *r* be the result of executing *credential*'s interface object's `[[Store]](credential, sameOriginWithAncestors)` internal method on *credential* and *sameOriginWithAncestors*.
    2. If *r* is an exception, reject *p* with *r*.

       Otherwise, resolve *p* with *r*.
6. Return *p*.

2.5.4. Create a `Credential`

The ***Create a `Credential`*** algorithm accepts a `CredentialCreationOptions` (*options*), and returns a `Promise` which resolves with a `Credential` if one can be created using the options provided, or `null` if no `Credential` can be created. In exceptional circumstances, the `Promise` may reject with an appropriate exception:

1. Let *settings* be the current settings object.
2. Assert: *settings* is a secure context.
3. Let *global* be *settings*' global object.
4. Let *sameOriginWithAncestors* be `true` if the current settings object is same-origin with its ancestors, and `false` otherwise.
5. Let *interfaces* be the set of *options*' relevant credential interface objects.
6. Return a promise rejected with `NotSupportedError` if any of the following statements are true:
    1. *settings* does not have a responsible document.
    2. *interfaces*' size is greater than 1.

Note: It may be reasonable at some point in the future to loosen this restriction, and allow the user agent to help the user choose among one of many potential credential types in order to support a "sign-up" use case. For the moment, though, we're punting on that by restricting the dictionary to a single entry.

7. If `options.signal`'s aborted flag is set, then return a promise rejected with an "`AbortError`" `DOMException`.
8. Let *p* be a new promise.
9. Let *origin* be *settings*'s origin.
10. Run the following steps in parallel:
    1. Let *r* be the result of executing *interfaces*[0]'s `[[Create]](origin, options, sameOriginWithAncestors)` internal method on *origin*, *options*, and *sameOriginWithAncestors*.
    2. If *r* is an exception, reject *p* with *r*, and terminate these substeps.
    3. If *r* is a `Credential` or `null`, resolve *p* with *r*, and terminate these substeps.
    4. Assert: *r* is a algorithm (as defined in §2.2.1.4 [[Create]] internal method).
    5. Queue a task on *global*'s DOM manipulation task source to run the following substeps:
        1. Resolve *p* with the result of promise-calling *r* given *global*.
11. Return *p*.

§ 2.5.5. Prevent Silent Access

The ***Prevent Silent Access*** algorithm accepts an environment settings object (*settings*), and returns a `Promise` which resolves once the `prevent silent access` flag is persisted to the credential store.

1. Let *origin* be *settings*' origin.

2. Let *p* be a new promise
3. Run the following seps in parallel:
    1. Set *origin*'s `prevent silent access` flag in the credential store.
    2. Resolve *p* with `undefined`.
4. Retun *p*.

§ 3. Password Credentials

For good or for ill, many websites rely on username/password pairs as an authentication mechanism. The `PasswordCredential` interface is a credential meant to enable this use case, storing both a username and password, as well as metadata that can help a user choose the right account from within a credential chooser.

§ 3.1. Examples

§ 3.1.1. Password-based Sign-in

EXAMPLE 5

MegaCorp, Inc. supports passwords, and can use `navigator.credentials.get()` to obtain username/password pairs from a user's credential store:

```
navigator.credentials
  .get({ 'password': true })
  .then(credential => {
    if (!credential) {
      // The user either doesn’t have credentials for this site, or
      // refused to share them. Insert some code here to fall back to
      // a basic login form.
      return;
    }
    if (credential.type == 'password') {
      var form = new FormData();
      form.append('username_field', credential.id);
      form.append('password_field', credential.password);
      var opt = {
        method: 'POST',
        body: form,
        credentials: 'include'  // Send cookies.
      };
      fetch('https://example.com/loginEndpoint', opt)
        .then(function (response) {
          if (/* |response| indicates a successful login */) {
            // Record that the credential was effective. See note below.
            navigator.credentials.store(credential);
            // Notify the user that sign-in succeeded! Do amazing, signed-in things!
            // Maybe navigate to a landing page via location.href =
            // '/signed-in-experience'?
          } else {
            // Insert some code here to fall back to a basic login form.
          }
        });
    }
  });
```

Alternatively, the website could just copy the credential data into a `<form>` and call `submit()` on the form:

```
navigator.credentials
  .get({ 'password': true })
  .then(credential => {
    if (!credential) {
      return; // as above...
    }
    if (credential.type === 'password') {
      document.querySelector('input[name=username_field]').value =
        credential.id;
      document.querySelector('input[name=password_field]').value =
        credential.password;
      document.getElementById('myform').submit();
    }
  });
```

Note that the former method is much preferred, as it contains an explicit call to `store()` and saves the credentials. The `<form>` based mechanism relies on form submission, which navigates the browsing context, making it difficult to ensure that `store()` is called after successful sign-in.

Note: The credential chooser presented by the user agent could allow the user to choose credentials that aren't actually stored for the current origin. For instance, it might offer up credentials from `https://m.example.com` when signing into `https://www.example.com` (as described in §6.1 Cross-domain credential access), or it might allow a user to create a new credential on the fly. Developers can deal gracefully with this uncertainty by calling `store()` every time credentials are successfully used, even right after credentials have been retrieved from `get()`: if the credentials aren't yet stored for the origin, the user will be given the opportunity to do so. If they are stored, the user won't be prompted.

§ 3.1.2. Post-sign-in Confirmation

To ensure that users are offered to store new credentials after a successful sign-in, they can to be passed to `store()`.

EXAMPLE 6

If a user is signed in by submitting the credentials to a sign-in endpoint via `fetch()`, we can check the response to determine whether the user was signed in successfully, and notify the user agent accordingly. Given a sign-in form like the following:

```
<form action="https://example.com/login" method="POST" id="theForm">
  <label for="username">Username</label>
  <input type="text" id="username" name="username" autocomplete="username">
  <label for="password">Password</label>
  <input type="password" id="password" name="password" autocomplete="current-password">
  <input type="submit">
</form>
```

Then the developer can handle the form submission with something like the following handler:

```
document.querySelector('#theForm').addEventListener('submit', e => {
    if (window.PasswordCredential) {
      e.preventDefault();

      // Construct a new PasswordCredential from the HTMLFormElement
      // that fired the "submit" event: this will suck up the values of the fields
      // labeled with "username" and "current-password" autocomplete
      // attributes:
      var c = new PasswordCredential(e.target);

      // Fetch the form’s action URL, passing that new credential object in
      // as a FormData object. If the response indicates success, tell the user agent
      // so it can ask the user to store the password for future use:
      var opt = {
        method: 'POST',
        body: new FormData(e.target),
        credentials: 'include'  // Send cookies.
      };
      fetch(e.target.action, opt).then(r => {
        if (/* |r| is a "successful" Response */)
          navigator.credentials.store(c);
      });
    }
});
```

3.1.3. Change Password

This same storage mechanism can be reused for "password change" with no modifications: if the user changes their credentials, the website can notify the user agent that they’ve successfully signed in with new credentials. The user agent can then update the credentials it stores:

EXAMPLE 7

MegaCorp Inc. allows users to change their passwords by POSTing data to a backend server asynchronously. After doing so successfully, they can update the user’s credentials by calling `store()` with the new information.

Given a password change form like the following:

```
<form action="https://example.com/changePassword" method="POST" id="theForm">
  <input type="hidden" name="username" autocomplete="username" value="user">
  <label for="password">New Password</label>
  <input type="password" id="password" name="password" autocomplete="new-password">
  <input type="submit">
</form>
```

The developer can handle the form submission with something like the following:

```
document.querySelector('#theForm').addEventListener('submit', e => {
  if (window.PasswordCredential) {
    e.preventDefault();

    // Construct a new PasswordCredential from the HTMLFormElement
    // that fired the "submit" event: this will suck up the values of the fields
    // labeled with "username" and "new-password" autocomplete
    // attributes:
    var c = new PasswordCredential(e.target);

    // Fetch the form’s action URL, passing that new credential object in
    // as a FormData object. If the response indicates success, tell the user agent
    // so it can ask the user to store the password for future use:
    var opt = {
      method: 'POST',
      body: new FormData(e.target),
      credentials: 'include'  // Send cookies.
    };
    fetch(e.target.action, opt).then(r => {
      if (/* |r| is a "successful" Response */)
        navigator.credentials.store(c);
    });
  }
});
```

§ 3.2. The `PasswordCredential` Interface

```
[Constructor(HTMLFormElement form),
 Constructor(PasswordCredentialData data),
 Exposed=Window,
 SecureContext]
interface PasswordCredential : Credential {
  readonly attribute USVString password;
};
PasswordCredential includes CredentialUserData;

partial dictionary CredentialRequestOptions {
  boolean password = false;
};
```

`password`, of type USVString, readonly
This attribute represents the password of the credential.

`[[type]]`
The `PasswordCredential` interface object has an internal slot named `[[type]]` whose value is "***`password`***".

`[[discovery]]`
The `PasswordCredential` interface object has an internal slot named `[[discovery]]` whose value is "`credential store`".

`PasswordCredential(form)`
This constructor accepts an `HTMLFormElement` (*form*), and runs the following steps:

1. Let *origin* be the current settings object's origin.
2. Let *r* be the result of executing Create a `PasswordCredential` from an `HTMLFormElement` given *form* and *origin*.
3. If *r* is an exception, throw *r*.

Otherwise, return *r*.

`PasswordCredential(data)`

This constructor accepts a `PasswordCredentialData` (*data*), and runs the following steps:

1. Let *r* be the result of executing Create a `PasswordCredential` from `PasswordCredentialData` on *data*.
2. If *r* is an exception, throw *r*.

   Otherwise, return *r*.

`PasswordCredential` objects can be created via `navigator.credentials.create()` either explicitly by passing in a `PasswordCredentialData` dictionary, or based on the contents of an `HTMLFormElement`'s submittable elements.

```
dictionary PasswordCredentialData : CredentialData {
  USVString name;
  USVString iconURL;
  required USVString origin;
  required USVString password;
};

typedef (PasswordCredentialData or HTMLFormElement) PasswordCredentialInit;

partial dictionary CredentialCreationOptions {
  PasswordCredentialInit password;
};
```

`PasswordCredential` objects are origin bound.

`PasswordCredential`'s interface object inherits `Credential`'s implementation of

`[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, and defines its own implementation of `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`, `[[Create]](origin, options, sameOriginWithAncestors)`, and `[[Store]](credential, sameOriginWithAncestors)`.

§ 3.3. Algorithms

§ 3.3.1. `PasswordCredential`'s `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`

`[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` is called with an origin (*origin*), a `CredentialRequestOptions` (*options*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*). The algorithm returns a set of `Credential` objects from the credential store. If no matching `Credential` objects are available, the returned set will be empty.

The algorithm will return a `NotAllowedError` if *sameOriginWithAncestors* is not `true`.

1. Assert: *options*["`password`"] exists.
2. If |sameOriginWithAncestors is `false`, return a "`NotAllowedError`" `DOMException`.

   Note: This restriction aims to address the concern raised in §6.4 Origin Confusion.

3. Return the empty set if *options*["`password`"] is not `true`.
4. Return the result of retrieving credentials from the credential store that match the following filter:
   1. The credential is a `PasswordCredential`
   2. The credential's `[[origin]]` is the same origin as *origin*.

§ 3.3.2. `PasswordCredential`'s `[[Create]](origin, options, sameOriginWithAncestors)`

`[[Create]](origin, options, sameOriginWithAncestors)` is called with an origin (*origin*), a `CredentialCreationOptions` (*options*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*). The algorithm returns a `PasswordCredential` if one can be created, `null` otherwise. The `CredentialCreationOptions` dictionary must have a `password` member which holds either an `HTMLFormElement` or a `PasswordCredentialData`. If that member's value cannot be used to create a `PasswordCredential`, this algorithm will return a `TypeError` exception.

1. Assert: *options*["`password`"] exists, and *sameOriginWithAncestors* is unused.
2. If *options*["`password`"] is an `HTMLFormElement`, return the result of executing Create a `PasswordCredential` from an `HTMLFormElement` given *options*["`password`"] and *origin*.
3. If *options*["`password`"] is a `PasswordCredentialData`, return the result of executing Create a `PasswordCredential` from `PasswordCredentialData` given *options*["`password`"].
4. Return a `TypeError` exception.

§ 3.3.3. `PasswordCredential`'s `[[Store]](credential, sameOriginWithAncestors)`

`[[Store]](credential, sameOriginWithAncestors)` is called with a `PasswordCredential` (*credential*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*). The algorithm returns `undefined` once *credential* is persisted to the credential store.

The algorithm will return a `NotAllowedError` if *sameOriginWithAncestors* is not `true`.

1. Return a "`NotAllowedError`" `DOMException` without altering the user agent's credential store if *sameOriginWithAncestors* is `false`.

> Note: This restriction aims to address the concern raised in §6.4 Origin Confusion.

2. If the user agent’s credential store contains a `PasswordCredential` (*stored*) whose `id` attribute is *credential*’s `id` and whose `[[origin]]` slot is the same origin as *credential*’s `[[origin]]`, then:

    1. If the user grants permission to update credentials (as discussed when defining user mediation), then:

        1. Set *stored*’s `password` to *credential*’s `password`.
        2. Set *stored*’s `name` to *credential*’s `name`.
        3. Set *stored*’s `iconURL` to *credential*’s `iconURL`.

    Otherwise, if the user grants permission to store credentials (as discussed when defining user mediation, then:

    1. Store a `PasswordCredential` in the credential store with the following properties:

        `id`
        : *credential*’s `id`

        `name`,
        : *credential*’s `name`

        `iconURL`
        : *credential*’s `iconURL`

        `[[origin]]`
        : *credential*’s `[[origin]]`

        `password`
        : *credential*’s `password`

3. Return `undefined`.

§ 3.3.4. Create a `PasswordCredential` from an `HTMLFormElement`

To ***Create a `PasswordCredential` from an `HTMLFormElement`***, given an `HTMLFormElement` (*form*) and an origin (*origin*), run these steps.

> Note: §3.1.2 Post-sign-in Confirmation and §3.1.3 Change Password provide examples of the intended usage.

1. Let *data* be a new `PasswordCredentialData` dictionary.
2. Set *data*'s `origin` member's value to *origin*'s value.
3. Let *formData* be the result of executing the `FormData` constructor on *form*.
4. Let *elements* be a list of all the submittable elements whose form owner is *form*, in tree order.
5. Let *newPasswordObserved* be `false`.
6. For each *field* in *elements*, run the following steps:
    1. If *field* does not have an `autocomplete` attribute, then skip to the next *field*.
    2. Let *name* be the value of *field*'s `name` attribute.
    3. If *formData*'s `has()` method returns `false` when executed on *name*, then skip to the next *field*.
    4. If *field*'s `autocomplete` attribute's value contains one or more autofill detail tokens (*tokens*), then:
        1. For each *token* in *tokens*:
            1. If *token* is an ASCII case-insensitive match for one of the following strings, run the associated steps:

               "**`new-password`**"

               Set *data*'s `password` member's value to the result of executing *formData*'s `get()` method on *name*, and *newPasswordObserved* to `true`.

               "**`current-password`**"

If *newPasswordObserved* is `false`, set *data*'s `password` member's value to the result of executing *formData*'s `get()` method on *name*.

> Note: By checking that *newPasswordObserved* is `false`, `new-password` fields take precedence over `current-password` fields.

"`photo`"
: Set *data*'s `iconURL` member's value to the result of executing *formData*'s `get()` method on *name*.

"`name`"
"`nickname`"
: Set *data*'s `name` member's value to the result of executing *formData*'s `get()` method on *name*.

"`username`"
: Set *data*'s `id` member's value to the result of executing *formData*'s `get()` method on *name*.

7. Let *c* be the result of executing Create a `PasswordCredential` from `PasswordCredentialData` on *data*.
8. If *c* is an exception, return *c*.
9. Assert: *c* is a `PasswordCredential`.
10. Return *c*.

§ 3.3.5. Create a `PasswordCredential` from `PasswordCredentialData`

To ***Create a `PasswordCredential` from `PasswordCredentialData`***, given an `PasswordCredentialData` (*data*), run these steps.

1. Let *c* be a new `PasswordCredential` object.

2. If any of the following are the empty string, return a `TypeError` exception:
    - *data*'s `id` member's value
    - *data*'s `origin` member's value
    - *data*'s `password` member's value
3. Set *c*'s properties as follows:

    `password`
    : *data*'s `password` member's value

    `id`
    : *data*'s `id` member's value

    `iconURL`
    : *data*'s `iconURL` member's value

    `name`
    : *data*'s `name` member's value

    `[[origin]]`
    : *data*'s `origin` member's value.
4. Return *c*.

§ 3.3.6. `CredentialRequestOptions` Matching for `PasswordCredential`

Given a `CredentialRequestOptions` (*options*), the following algorithm returns "`Matches`" if the `PasswordCredential` should be available as a response to a `get()` request, and "`Does Not Match`" otherwise.

1. If *options* has a `password` member whose value is `true`, then return "`Matches`".

2. Return "`Does Not Match`".

§ 4. Federated Credentials

§ 4.1. The `FederatedCredential` Interface

```
[Constructor(FederatedCredentialInit data),
 Exposed=Window,
 SecureContext]
interface FederatedCredential : Credential {
  readonly attribute USVString provider;
  readonly attribute DOMString? protocol;
};
FederatedCredential includes CredentialUserData;

dictionary FederatedCredentialRequestOptions {
  sequence<USVString> providers;
  sequence<DOMString> protocols;
};

partial dictionary CredentialRequestOptions {
  FederatedCredentialRequestOptions federated;
};
```

`provider`, of type USVString, readonly

The credential's federated identity provider. See §4.1.1 Identifying Providers for details regarding valid formats.

`protocol`, of type DOMString, readonly, nullable

The credential's federated identity provider's protocol (e.g. "`openidconnect`"). If the value is `null`, then the protocol can be inferred from the `provider`.

`[[type]]`
The `FederatedCredential` interface object has an internal slot named `[[type]]` whose value is "***federated***".

`[[discovery]]`
The `FederatedCredential` interface object has an internal slot named `[[discovery]]` whose value is "`credential store`".

`FederatedCredential(data)`
This constructor accepts a `FederatedCredentialInit` (*data*), and runs the following steps:

1. Let *r* be the result of executing Create a `FederatedCredential` from `FederatedCredentialInit` on *data*.
2. If *r* is an exception, throw *r*.

   Otherwise, return *r*.

`FederatedCredential` objects can be created by passing a `FederatedCredentialInit` dictionary into `navigator.credentials.create()`.

```
dictionary FederatedCredentialInit : CredentialData {
  USVString name;
  USVString iconURL;
  required USVString origin;
  required USVString provider;
  DOMString protocol;
};

partial dictionary CredentialCreationOptions {
  FederatedCredentialInit federated;
};
```

`FederatedCredential` objects are origin bound.

`FederatedCredential`'s interface object inherits `Credential`'s implementation of `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, and defines its own implementation of `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`, `[[Create]](origin, options, sameOriginWithAncestors)`, and `[[Store]](credential, sameOriginWithAncestors)`.

Note: If, in the future, we teach the user agent to obtain authentication tokens on a user's behalf, we could do so by building an implementation of `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`.

4.1.1. Identifying Providers

Every site should use the same identifier when referring to a specific federated identity provider. For example, Facebook Login shouldn't be referred to as "Facebook" and "Facebook Login" and "FB" and "FBL" and "Facebook.com" and so on. It should have a canonical identifier which everyone can make use of, as consistent identification makes it possible for user agents to be helpful.

For consistency, federations passed into the APIs defined in this document (e.g. `FederatedCredentialRequestOptions`'s `providers` array, or `FederatedCredential`'s `provider` property) MUST be identified by the ASCII serialization of the origin the provider uses for sign in. That is, Facebook would be represented by `https://www.facebook.com` and Google by `https://accounts.google.com`.

This serialization of an origin does _not_ include a trailing U+002F SOLIDUS ("`/`"), but user agents SHOULD accept them silently: `https://accounts.google.com/` is clearly intended to be the same as `https://accounts.google.com`.

§ 4.2. Algorithms

§ 4.2.1. `FederatedCredential`'s `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`

`[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` is called with an origin (*origin*), a `CredentialRequestOptions` (*options*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*). The algorithm returns a set of `Credential` objects from the credential store. If no matching `Credential` objects are available, the returned set will be empty.

1. Assert: *options*["`federated`"] exists.
2. If *sameOriginWithAncestors* is `false`, return a "`NotAllowedError`" `DOMException`.

   Note: This restriction aims to address the concern raised in §6.4 Origin Confusion.

3. Return the empty set if *options*["`federated`"] is not `true`.
4. Return the result of retrieving credentials from the credential store that match the following filter:
   1. The credential is a `FederatedCredential`
   2. The credential's `[[origin]]` is the same origin as *origin*.
   3. If *options*["`federated`"]["`providers`"] exists, its value contains the credentials's `provider`.
   4. If *options*["`federated`"]["`protocols`"] exists, its value contains the credentials's `protocol`.

§ 4.2.2. `FederatedCredential`'s `[[Create]](origin, options, sameOriginWithAncestors)`

`[[Create]](origin, options, sameOriginWithAncestors)` is called with an origin (*origin*), a `CredentialCreationOptions` (*options*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*). The algorithm returns a `FederatedCredential` if one can be created, `null` otherwise, or an exception in exceptional circumstances:

1. Assert: *options*["`federated`"] exists, and *sameOriginWithAncestors* is unused.
2. Set *options*["`federated`"]'s `origin` member's value to *origin*'s value.
3. Return the result of executing Create a `FederatedCredential` from `FederatedCredentialInit` given *options*["`federated`"].

4.2.3. `FederatedCredential`'s `[[Store]](credential, sameOriginWithAncestors)`

`[[Store]](credential, sameOriginWithAncestors)` is called with a `FederatedCredential` (*credential*), and a boolean which is `true` iff the calling context is same-origin with its ancestors (*sameOriginWithAncestors*). The algorithm returns `undefined` once *credential* is persisted to the credential store.

The algorithm will return a `NotAllowedError` if *sameOriginWithAncestors* is not `true`.

1. Return a "`NotAllowedError`" `DOMException` without altering the user agent's credential store if *sameOriginWithAncestors* is `false`.

   Note: This restriction aims to address the concern raised in §6.4 Origin Confusion.

2. If the user agent's credential store contains a `FederatedCredential` whose `id` attribute is *credential*'s `id` and whose `[[origin]]` slot is the same origin as *credential*'s `[[origin]]`, and whose `provider` is *credential*'s `provider`, then return.
3. If the user grants permission to store credentials (as discussed when defining user mediation), then store a `FederatedCredential` in the credential store with the following properties:

   `id`

credential’s `id`

`name`,
credential’s `name`

`iconURL`
credential’s `iconURL`

`[[origin]]`
credential’s `[[origin]]`

`provider`
credential’s `provider`

`protocol`
credential’s `protocol`

4. Return `undefined`.

4.2.4. Create a `FederatedCredential` from `FederatedCredentialInit`

To ***Create a `FederatedCredential` from `FederatedCredentialInit`***, given a `FederatedCredentialInit` (*init*), run these steps.

1. Let *c* be a new `FederatedCredential` object.
2. If any of the following are the empty string, return a `TypeError` exception:
   - *init*.`id`'s value
   - *init*.`provider`'s value
3. Set *c*’s properties as follows:

`id`

init.`id`'s value

`provider`
: *init*.`provider`'s value

`iconURL`
: *init*.`iconURL`'s value

`name`
: *init*.`name`'s value

`[[origin]]`
: *init*.`origin`'s value.

4. Return *c*.

§ 5. User Mediation

Exposing credential information to the web via an API has a number of potential impacts on user privacy. The user agent, therefore, MUST involve the user in a number of cases in order to ensure that they clearly understands what's going on, and with whom their credentials are being shared.

We call a particular action ***user mediated*** if it takes place after gaining a user's explicit consent. Consent might be expressed through a user's direct interaction with a credential chooser interface, for example. In general, user mediated actions will involve presenting the user some sort of UI, and asking them to make a decision.

An action is unmediated if it takes place silently, without explicit user consent. For example, if a user configures their browser to grant persistent credential access to a particular origin, credentials may be provided without presenting the user with a UI requesting a decision.

Here we'll spell out a few requirements that hold for all credential types, but note that there's a good deal of latitude left up to the user agent (which is in a priviliged position to assist the user). Moreover, specific credential types may have distinct requirements that exceed the requirements laid out more generally here.

§ 5.1. Storing and Updating Credentials

Credential information is sensitive data, and users MUST remain in control of that information's storage. Inadvertent credential storage could, for instance, unexpectedly link a user's local profile on a particular device to a specific online persona. To mitigate the risk of surprise:

1. Credential information SHOULD NOT be stored or updated without user mediation. For example, the user agent could display a "Save this credential?" dialog box to the user in response to each call to `store()`.

   User consent MAY be inferred if a user agent chooses to offer a persistant grant of consent in the form of an "Always save passwords" option (though we'd suggest that user agents should err on the side of something more narrowly scoped: perhaps "Always save _generated_ passwords.", or "Always save passwords for this site.").
2. User agents SHOULD notify users when credentials are stored. This might take the form of an icon in the address bar, or some similar location.
3. User agents MUST allow users to manually remove stored credentials. This functionality might be implemented as a settings page, or via interaction with a notification as described above.

§ 5.2. Requiring User Mediation

By default, user mediation is required for all origins, as the relevant prevent silent access flag in the credential store is set to `true`. Users MAY choose to grant an origin persistent access to credentials (perhaps in the form of a "Stay signed into this site." option), which would set this flag to `false`. In this case, the user would always be signed into that site, which is desirable from the perspective of usability and convinience, but which might nevertheless have surprising implications (consider a user agent which syncs this flag's state across devices, for instance).

To mitigate the risk of surprise:

1. User agents MUST allow users to require user mediation for a given origin or for all origins. This functionality might be implemented as a global toggle that overrides each origin's `prevent silent access` flag to return `false`, or via more granular settings for specific origins (or specific credentials on specific origins).
2. User agents MUST NOT set an origin's `prevent silent access` flag to `false` without user mediation. For example, the credential chooser described in §5.3 Credential Selection could have a checkbox which the user could toggle to mark a credential as available without mediation for the origin, or the user agent could have an onboarding process for its credential manager which asked a user for a default setting.
3. User agents MUST notify users when credentials are provided to an origin. This could take the form of an icon in the address bar, or some similar location.
4. If a user clears her browsing data for an origin (cookies, localStorage, and so on), the user agent MUST set the `prevent silent access` flag to `true` for that origin.

§ 5.3. Credential Selection

When responding to a call to `get()` on an origin which requires user mediation, user agents MUST ask the user for permission to share credential information. This SHOULD take the form of a ***credential chooser*** which presents the user with a list of credentials that are available for use on a site, allowing them to select one which should be provided to the website, or to reject the request entirely.

The chooser interface SHOULD be implemented in such a way as to be distinguishable from UI which a website could produce. For example, the chooser might overlap the user agent's UI in some unspoofable way.

The chooser interface MUST include an indication of the origin which is requesting credentials.

The chooser interface SHOULD include all `Credential` objects associated with the origin that requested credentials.

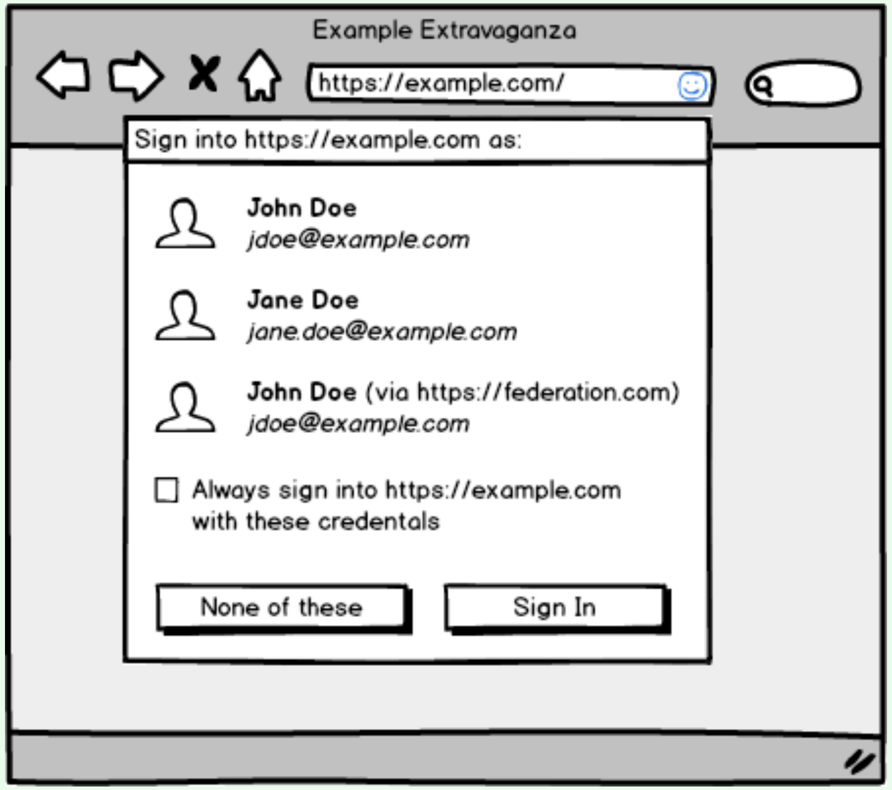
User agents MAY internally associate information with each `Credential` object beyond the attributes specified in this document in order to enhance the utility of such a chooser. For example, favicons could help disambiguate identity providers, etc. Any additional information stored MUST not be exposed directly to the web.

The chooser's behavior is not defined here: user agents are encouraged to experiment with UI treatments that educate users about their authentication options, and guide them through the process of choosing a credential to present. That said, the interface to the chooser is as follows:

The user agent can ***ask the user to choose a `Credential`***, given a `CredentialRequestOptions` (*options*), and a set of `Credential` objects from the credential store (*locally discovered credentials*).

This algorithm returns either `null` if the user chose not to share a credential with the site, a `Credential` object if the user chose a specific credential, or a `Credential` interface object if the user chose a type of credential.

It seems reasonable for the chooser interface to display the list of *locally discovered credentials* to the user, perhaps something like this exceptionally non-normative mock:


Example Extravaganza
https://example.com/
Sign into https://example.com as:
John Doe
jdoe@example.com
Jane Doe
jane.doe@example.com
John Doe (via https://federation.com)
jdoe@example.com
Always sign into https://example.com with these credentals
None of these
Sign In

If the *options* provided is not matchable a priori, then it might also make sense for the chooser interface to list the relevant credential interface objects for *options* that aren't covered by the list of explicit credentials. If, for instance, a site accepts webauthn-style authenticators, then "Security Key" might show up in the chooser list with an appropriate icon.

Also, note that in some cases the user agent may skip the chooser entirely. For example, if the only relevant credential interface objects is one that itself requires user interaction, the user agent may return that interface directly, and rely on its internal mediation flow for user consent.

§ 6. Security and Privacy Considerations

The following sections represent guidelines for various security and privacy considerations. Individual credential types may enforce stricter or more relaxed versions of these guidelines.

§ 6.1. Cross-domain credential access

Credentials are sensitive information, and user agents need to exercise caution in determining when they can be safely shared with a website. The safest option is to restrict credential sharing to the exact origin on which they were saved. That is likely too restrictive for the web, however: consider sites which divide functionality into subdomains like `example.com` vs `admin.example.com`.

As a compromise between annoying users, and securing their credentials, user agents:

1. MUST NOT share credentials between origins whose scheme components represent a downgrade in security. That is, it may make sense to allow credentials saved on `http://example.com/` to be made available to `https://example.com/` (in order to encourage developers to migrate to secure transport), but the inverse would be dangerous.
2. MAY use the Public Suffix List [PSL] to determine the effective scope of a credential by comparing the registerable domain of the credential's `[[origin]]` with the origin in which `get()` is called. That is: credentials saved on `https://admin.example.com/` and `https://example.com/` MAY be offered to users when `get()` is called from `https://www.example.com/`, and vice versa.
3. MUST NOT offer credentials to an origin in response to `get()` without user mediation if the credential's origin is not an exact match for the calling origin. That is, `Credential` objects for `https://example.com` would not be returned directly to `https://www.example.com`, but could be offered to the user via the chooser.

§ 6.2. Credential Leakage

Developers are well-advised to take some precautions to mitigate the risk that a cross-site scripting attack could turn into persistent access to a user's account by setting a reasonable Content Security Policy [CSP] which restricts the endpoints to which data can be sent. In particular, developers should ensure that the following directives are set, explicitly or implicitly, in their pages' policies:

- `script-src` and `object-src` both restrict script execution on a page, making it less likely that a cross-site scripting attack will succeed in the first place. If sites are populating `<form>` elements, also `form-action` directives should be set.
- `connect-src` restricts the origins to which `fetch()` may submit data (which mitigates the risk that credentials could be exfiltrated to `evil.com`.
- `child-src` restricts the nested browsing contexts which may be embedded in a page, making it more difficult to inject a malicious `postMessage()` target. [WEBMESSAGING]

Developers should, of course, also properly escape input and output, and consider using other layers of defense, such as Subresource Integrity [SRI] to further reduce risk.

When defining specific credential types, specific credential types SHOULD give due consideration to the ways in which credential data can be transmitted over the wire. It might be reasonable, for example, to define transmission mechanisms which are restricted to same-origin endpoints.

§ 6.3. Insecure Sites

User agents MUST NOT expose the APIs defined here to environments which are not secure contexts. User agents might implement autofill mechanisms which store user credentials and fill sign-in forms on non-a priori authenticated URLs, but those sites cannot be trusted to interact directly with the credential manager in any meaningful way, and those sites MUST NOT have access to credentials saved in secure contexts.

§ 6.4. Origin Confusion

If framed pages have access to the APIs defined here, it might be possible to confuse a user into granting access to credentials for an origin other than the top-level browsing context, which is the only security origin which users can reasonably be expected to understand.

This document exposes the Credential Management APIs to those contexts, as it's likely that some credential types will be straightforward to make available if user agents put enough thought and context into their UI.

Specific credential types, however, will be difficult to expose in those contexts without risk. Those credential types are restricted via checks in their `[[Create]](origin, options, sameOriginWithAncestors)`, `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`, `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, and `[[Store]](credential, sameOriginWithAncestors)` methods, as appropriate.

For example `PasswordCredential`'s `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` method will immedietely return an empty set if called from inside a `Worker`, or a non-top-level browsing context.

§ 6.5. Timing Attacks

If the user has no credentials for an origin, a call to `get()` will resolve very quickly indeed. A malicious website could distinguish between a user with no credentials and a user with credentials who chooses not to share them.

User agents SHOULD also rate-limit credential requests. It's almost certainly abusive for a page to request credentials more than a few times in a short period.

§ 6.6. Signing-Out

If a user has chosen to automatically sign-in to websites, as discussed in §5.2 Requiring User Mediation, then the user agent will provide credentials to an origin whenever it asks for them. The website can instruct the user agent to suppress this behavior by calling `CredentialsContainer`'s `preventSilentAccess()` method, which will turn off automatic sign-in for a given origin.

The user agent relies on the website to do the right thing; an inattentive (or malicious) website could simply neglect to call this method, causing the user agent to continue providing credentials against the user’s apparent intention. This is marginally worse than the status-quo of a site that doesn’t clear user credentials when they click "Sign-out", as the user agent becomes complicit in the authentication.

The user MUST have some control over this behavior. As noted in §5.2 Requiring User Mediation, clearing cookies for an origin will also reset that origin’s `prevent silent access` flag the credential store to `true`. Additionally, the user agent SHOULD provide some UI affordance for disabling automatic sign-in for a particular origin. This could be tied to the notification that credentials have been provided to an origin, for example.

§ 6.7. Chooser Leakage

If a user agent’s credential chooser displays images supplied by an origin (for example, if a `Credential` displays a site’s favicon), then, requests for these images MUST NOT be directly tied to instantiating the chooser in order to avoid leaking chooser usage. One option would be to fetch the images in the background when saving or updating a `Credential`, and to cache them for the lifetime of the `Credential`.

These images MUST be fetched with the credentials mode set to "`omit`", the service-workers mode set to "`none`", the client set to `null`, the initiator set to the empty string, and the destination "`subresource`".

Moreover, if the user agent allows the user to change either the name or icon associated with the credential, the alterations to the data SHOULD NOT be exposed to the website (consider a user who names two credentials for an origin "My fake account" and "My real account", for instance).

§ 6.8. Locally Stored Data

This API offers an origin the ability to store data persistently along with a user's profile. Since most user agents treat credential data differently than "browsing data" (cookies, etc.) this might have the side effect of surprising a user who might believe that all traces of an origin have been wiped out when they clear their cookies.

User agents SHOULD provide UI that makes it clear to a user that credential data is stored for an origin, and SHOULD make it easy for users to remove such data when they're no longer interested in keeping it around.

§ 7. Implementation Considerations

This section is non-normative.

§ 7.1. Website Authors

ISSUE 4 Add some thoughts here about when and how the API should be used, especially with regard to `mediation`. <https://github.com/w3c/webappsec/issues/290>

ISSUE 5 Describe encoding restrictions of submitting credentials by `fetch()` with a `FormData` body.

When performing feature detection for a given credential type, developers are encouraged to verify that the relevant `Credential` specialization is present, rather than relying on the presence of `navigator.credentals`. The latter verifies the existence of the API itself, but does not ensure that the specific kind of credential necessary for a given site is supported. For example, if a given site requires passwords, checking `if (window.PasswordCredential)` is the most effective verification of support.

§ 7.2. Extension Points

This document provides a generic, high-level API that's meant to be extended with specific types of credentials that serve specific authentication needs. Doing so is, hopefully, straightforward:

1. Define a new interface that inherits from `Credential`:

   EXAMPLE 8

   ```
   [Exposed=Window,
    SecureContext]
   interface ExampleCredential : Credential {
     // Definition goes here.
   };
   ```

2. Define appropriate `[[Create]](origin, options, sameOriginWithAncestors)`, `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)`, `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)`, and `[[Store]](credential, sameOriginWithAncestors)` methods on `ExampleCredential`'s interface object. `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` is appropriate for credentials that remain effective forever and can therefore simply be copied out of the credential store, while `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` is appropriate for credentials that need to be re-generated from a credential source.

   Long-running operations, like those in `PublicKeyCredential`'s `[[Create]](origin, options, sameOriginWithAncestors)` and `[[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)` operations are encouraged to use `options.signal` to allow developers to abort the operation. See DOM §3.3 Using AbortController and AbortSignal objects in APIs for detailed instructions.

EXAMPLE 9

`ExampleCredential`'s `[[CollectFromCredentialStore]](origin, options, sameOriginWithAncestors)` internal method is called with an origin (`origin`), a CredentialRequestOptions object (`options`), and a boolean which is `true` iff the calling context is same-origin with its ancestors. The algorithm returns a set of `Credential` objects that match the options provided. If no matching `Credential` objects are available, the returned set will be empty.

1. Assert: `options[example]` exists.
2. If `options[example]` is not truthy, return the empty set.
3. For each *credential* in the credential store:
    1. ...

3. Define the value of the `ExampleCredential` interface object's `[[type]]` slot:

EXAMPLE 10

The `ExampleCredential` interface object has an internal slot named `[[type]]` whose value is the string `"example"`.

4. Define the value of the `ExampleCredential` interface object's `[[discovery]]` slot:

EXAMPLE 11

The `ExampleCredential` interface object has an internal slot named `[[type]]` whose value is "`credential store`".

5. Extend `CredentialRequestOptions` with the options the new credential type needs to respond reasonably to `get()`:

EXAMPLE 12

```
dictionary ExampleCredentialRequestOptions {
  // Definition goes here.
};

partial dictionary CredentialRequestOptions {
  ExampleCredentialRequestOptions example;
};
```

6. Extend `CredentialCreationOptions` with the data the new credential type needs to create `Credential` objects in response to `create()`:

EXAMPLE 13

```
dictionary ExampleCredentialInit {
  // Definition goes here.
};

partial dictionary CredentialCreationOptions {
  ExampleCredentialInit example;
};
```

You might also find that new primitives are necessary. For instance, you might want to return many `Credential` objects rather than just one in some sort of complicated, multi-factor sign-in process. That might be accomplished in a generic fashion by adding a `getAll()` method to `CredentialsContainer` which returned a `sequence<Credential>`, and defining a reasonable mechanism for dealing with requesting credentials of distinct types.

For any such extension, we recommend getting in touch with public-webappsec@ for consultation and review.

§ 7.3. Browser Extensions

Ideally, user agents that implement an extension system of some sort will allow third-parties to hook into these API endpoints in order to improve the behavior of third party credential management software in the same way that user agents can improve their own via this imperative approach.

This could range from a complex new API that the user agent mediates, or simply by allowing extensions to overwrite the `get()` and `store()` endpoints for their own purposes.

§ 8. Future Work

This section is non-normative.

The API defined here does the bare minimum to expose user agent's credential managers to the web, and allows the web to help those credential managers understand when federated identity providers are in use. The next logical step will be along the lines sketched in documents like [WEB-LOGIN] (and, to some extent, Mozilla's BrowserID [BROWSERID]).

The user agent is in the unique position of being able to effectively mediate the relationship between users, identity providers, and websites. If the user agent can remove some of the risk and confusion associated with the typical authentication flows, users will be in a significantly better position than today.

A natural way to expose this information might be to extend the `FederatedCredential` interface with properties like authentication tokens, and possibly to add some form of manifest format with properties that declare the authentication type which the provider supports.

The API described here is designed to be extensible enough to support use cases that require user interaction, perhaps with websites other than the one which requested credentials. We hope that the Promise-based system we've settled on is extensible enough to support these kinds of asynchronous flows which could require some level of interaction between multiple browsing contexts (e.g. mediated activity on `idp.com` might resolve a Promise handed back to `rp.com`) or between devices and user agents (e.g. [WEBAUTHN]) in the future without redesigning the API from the ground up.

Baby steps.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

EXAMPLE 14

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

§ Conformant Algorithms

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Conformance requirements phrased as algorithms or specific steps can be implemented in any manner, so long as the end result is equivalent. In particular, the algorithms defined in this specification are intended to be easy to understand and are not intended to be performant. Implementers are encouraged to optimize.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[CSP] defines the following terms:
- child-src
- connect-src
- form-action
- object-src
- script-src

[DOM] defines the following terms:
- AbortSignal
- aborted flag
- context object
- tree order

[ecma262] defines the following terms:

- internal method

[FETCH] defines the following terms:

- Response
- client
- credentials mode
- destination
- fetch(input)
- initiator
- service-workers mode

[HTML] defines the following terms:

- HTMLFormElement
- Navigator
- Worker
- active document
- ascii serialization of an origin
- autocomplete
- autofill detail tokens
- current settings object
- current-password
- dom manipulation task source
- environment settings object
- form
- form owner
- global object
- in parallel
- name (for input)
- new-password
- nickname
- origin (for environment settings object)
- parent browsing context
- photo
- queue a task
- relevant realm
- responsible browsing context
- responsible document
- same origin
- submit()
- submittable elements
- task
- top-level browsing context
- username

[INFRA] defines the following terms:

- append
- ascii case-insensitive
- contain
- continue
- exist
- set
- size

[mixed-content] defines the following terms:

- a priori authenticated url

[promises-guide] defines the following terms:

- a new promise
- a promise rejected with
- promise-calling
- reject
- resolve

[PSL] defines the following terms:

- registerable domain

[secure-contexts] defines the following terms:

- secure contexts

[WEBAUTHN] defines the following terms:

- PublicKeyCredential
- [[Create]](origin, options, sameOriginWithAncestors)
- [[DiscoverFromExternalSource]](origin, options, sameOriginWithAncestors)

[WebIDL] defines the following terms:

- AbortError
- DOMException
- DOMString
- Exposed
- NotAllowedError
- Promise
- SameObject
- SecureContext
- TypeError
- USVString
- boolean
- exception
- inherit
- inherited interfaces
- interface object
- interface prototype object
- throw

[XHR] defines the following terms:

- FormData
- XMLHttpRequest
- get(name)
- has(name)

§ References

§ Normative References

[CSP]
Mike West. Content Security Policy Level 3. 13 September 2016. WD. URL: https://www.w3.org/TR/CSP3/

[DOM]
Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[FETCH]
Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[HTML]
Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]

Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[MIXED-CONTENT]
Mike West. Mixed Content. 2 August 2016. CR. URL: https://www.w3.org/TR/mixed-content/

[PROMISES-GUIDE]
Domenic Denicola. Writing Promise-Using Specifications. 16 February 2016. Finding of the W3C TAG. URL: https://www.w3.org/2001/tag/doc/promises-guide

[PSL]
Public Suffix List. Mozilla Foundation.

[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

[SECURE-CONTEXTS]
Mike West. Secure Contexts. 15 September 2016. CR. URL: https://www.w3.org/TR/secure-contexts/

[WebIDL]
Cameron McCormack; Boris Zbarsky; Tobie Langel. Web IDL. 15 December 2016. ED. URL: https://heycam.github.io/webidl/

[XHR]
Anne van Kesteren. XMLHttpRequest Standard. Living Standard. URL: https://xhr.spec.whatwg.org/

§ Informative References

[BROWSERID]
Ben Adida; et al. BrowserID. 26 February 2013. URL: https://github.com/mozilla/id-specs/blob/prod/browserid/index.md

[SRI]
Devdatta Akhawe; et al. Subresource Integrity. 23 June 2016. REC. URL: https://www.w3.org/TR/SRI/

[WEB-LOGIN]

Jason Denizac; Robin Berjon; Anne van Kesteren. web-login. URL: https://github.com/jden/web-login

[WEBAUTHN]
Dirk Balfanz; et al. Web Authentication: An API for accessing Public Key Credentials Level 1. 17 January 2019. PR. URL: https://www.w3.org/TR/webauthn/

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

[XMLHTTPREQUEST]
Anne van Kesteren; et al. XMLHttpRequest Level 1. 6 October 2016. NOTE. URL: https://www.w3.org/TR/XMLHttpRequest/

§ IDL Index

```
[Exposed=Window, SecureContext]
interface Credential {
  readonly attribute USVString id;
  readonly attribute DOMString type;
};

[SecureContext]
interface mixin CredentialUserData {
  readonly attribute USVString name;
  readonly attribute USVString iconURL;
};

partial interface Navigator {
  [SecureContext, SameObject] readonly attribute CredentialsContainer credentials;
};

[Exposed=Window, SecureContext]
interface CredentialsContainer {
  Promise<Credential?> get(optional CredentialRequestOptions options);
  Promise<Credential> store(Credential credential);
  Promise<Credential?> create(optional CredentialCreationOptions options);
  Promise<void> preventSilentAccess();
};

dictionary CredentialData {
  required USVString id;
};

dictionary CredentialRequestOptions {
  CredentialMediationRequirement mediation = "optional";
  AbortSignal signal;
};

enum CredentialMediationRequirement {
  "silent",
  "optional",
  "required"
};

dictionary CredentialCreationOptions {
  AbortSignal signal;
};

[Constructor(HTMLFormElement form),
 Constructor(PasswordCredentialData data),
 Exposed=Window,
 SecureContext]
interface PasswordCredential : Credential {
  readonly attribute USVString password;
};
PasswordCredential includes CredentialUserData;

partial dictionary CredentialRequestOptions {
  boolean password = false;
};

dictionary PasswordCredentialData : CredentialData {
  USVString name;
  USVString iconURL;
  required USVString origin;
  required USVString password;
```

```
};

typedef (PasswordCredentialData or HTMLFormElement) PasswordCredentialInit;

partial dictionary CredentialCreationOptions {
  PasswordCredentialInit password;
};

[Constructor(FederatedCredentialInit data),
 Exposed=Window,
 SecureContext]
interface FederatedCredential : Credential {
  readonly attribute USVString provider;
  readonly attribute DOMString? protocol;
};
FederatedCredential includes CredentialUserData;

dictionary FederatedCredentialRequestOptions {
  sequence<USVString> providers;
  sequence<DOMString> protocols;
};

partial dictionary CredentialRequestOptions {
  FederatedCredentialRequestOptions federated;
};

dictionary FederatedCredentialInit : CredentialData {
  USVString name;
  USVString iconURL;
  required USVString origin;
  required USVString provider;
  DOMString protocol;
};

partial dictionary CredentialCreationOptions {
  FederatedCredentialInit federated;
};
```

§ Issues Index

ISSUE 1 Talk to Tobie/Dominic about the interface object bits, here and in §2.5.1 Request a Credential, etc. I'm not sure I've gotten the terminology right. interface prototype object, maybe? ↵

ISSUE 2 jyasskin@ suggests replacing the iteration through the interface objects with a registry. I'm not sure which is less clear, honestly. I'll leave it like this for the moment, and we can argue about whether this is too much of a `COMEFROM` interface. ↵

ISSUE 3 This might be the wrong model. It would be nice to support a site that wished to accept either username/passwords or webauthn-style credentials without forcing a chooser for those users who use the former, and who wish to remain signed in. ↵

ISSUE 4 Add some thoughts here about when and how the API should be used, especially with regard to `mediation`. <https://github.com/w3c/webappsec/issues/290> ↵

ISSUE 5 Describe encoding restrictions of submitting credentials by `fetch()` with a `FormData` body. ↵