AdChoices User Preferences API Specification

Version 1

Digital Advertising Alliance
Feb 17, 2023

Table of Contents

Introduction and Scope

The purpose of the AdChoices-based Self-Regulatory Programs across the world is to promote enhanced transparency (generally speaking, data collection, use and disclosure notifications that sit outside of the privacy policy) and consumer control, as well as other elements of the DAA-family Self-Regulatory Frameworks (regardless of geographic region) related to the collection and use of data for Interest Based Advertising (IBA), sometimes referred to as online behavioral advertising (OBA).

This specification defines a standard JavaScript API which enables ad-serving code running on a site or app to discover IBA preferences for the user that can be known by DAA Recognized CMPs. It draws on patterns that may be familiar to implementers introduced in other industry-standard APIs such as the IAB's MRAID and CMP APIs.[1]

In a web environment, CMPs provide an implementation of the JavaScript API described in this document. Typically, this would be included in the typical script payload a publisher installs on their site. In an app environment, it is typically the responsibility of advertising SDKs to provide the API implementation, retrieving the IBA preferences in the manner described in "In-App Details" below.

The term "User Preferences API implementer" will be used in this document to refer to the CMP or the advertising SDK as is appropriate for the environment, with the terms "CMP" and "ad SDK" used where that party specifically is responsible.

This specification works in concert with the AdChoices String Specification; it presents the same data via a JavaScript API. Accordingly, this specification will reference the AdChoices String Specification frequently; refer to it for context and details. In other words, this specification should be viewed as a companion document to the AdChoices String Specification.

The primary intended audience for this specification is DAA Recognized CMPs and advertising SDK companies (for in-app environments). It is specifically intended for a technical audience (e.g., product managers, engineers, etc.).

[1] Some patterns and examples are adapted from the Transparency and Consent Framework Consent Management Platform API v2 from IAB Tech Lab, which is licensed under the Creative Commons Attribution 3.0 License. The API interface and message payload is similar to that specification, but modified to reflect the needs for AdChoices user preferences.

API Specification

AdChoices Function

User Preferences API implementers must provide a global function on the page or app named `daaGetAdChoices`. This function must be present before any ad serving code loads. It can initially be a stub of some sort, but any calls to it must eventually be queued and eventually serviced (i.e. callback called) after the full code has been loaded and after the stored AdChoices information has been retrieved, if any.

The function has the following form:

```
daaGetAdChoices(callback)
```

The caller provides a callback function which will be called asynchronously and takes a single argument: an AdChoices object (described below).

AdChoices Object

This object wraps the actual user preferences payload and describes the outcome of the User Preferences API Implementer's attempt to retrieve AdChoices preferences.

Key	**Type**	**Description**
success	Boolean	True if the implementer could retrieve stored AdChoices information for the user, otherwise false.
userPreferences	Object; optional	A user preferences object, see below. Only present if AdChoices information could be retrieved.

User Preferences Object

This object describes the user's preferences, if available. It contains an AdChoices string as defined by a consumer using the DAA Token and/or Category tools, as well as the same data presented in structured form for ease of access:

Key	**Type**	**Description**
adChoicesString	String	The AdChoices string for the user; see AdChoices String Specification.
version	Number	Version of preferences information; current version is 1.

timestamp	Number	The timestamp at which the preferences was created as seconds since Unix epoch.
globalChoice	Number	The user's global IBA choice; see Appendix 3 of the AdChoices String Specification.
participants	Array of Participant objects	An array of Participant objects, describing the user's per-participant choices. This will be an empty array when a global status other than 0 has been provided, since the global status applies to all, and providing per-participant statuses would give no useful additional information. The array will not contain objects for participants when their choice status is 0.
categories	Array of Category objects	An array of Category objects, describing the user's per-category choices. Objects should not be included for categories with a choice status of 0 since this provides no useful information. In the event that there are no category preferences, the array will be empty.

Participant Object

Key	**Type**	**Description**
participantId	Number	The ID of the participant described; see Appendix 1 of the AdChoices String Specification for how to retrieve metadata about DAA participants integrated into the relevant industry tools.
choice	Number	The user's choice for this participant; see Appendix 3 of the AdChoices String Specification.

Category Object

Key	**Type**	**Description**
categoryId	Number	The ID of the category described; see Appendix 2 of the AdChoices String Specification for how to retrieve metadata about categories.
preference	Number	The user's preference for this category; see Appendix 4 of the AdChoices String Specification.

Examples

These examples align with and match the numbered examples in the AdChoices String Specification.

Example 1: No Global Preference, Three Participant Preferences, One Category Preference

```
{
  success: true,
  userPreferences: {
    adChoicesString: "BYVHiWQADABEAIQAyABBIEA",
    version: 1,
    timestamp: 1632756313,
    globalChoice: 0,
    participants: [
      {
        participantId: 1,
        choice: 1
      },
      {
        participantId: 2,
        choice: 1
      },
      {
        participantId: 3,
        choice: 2
      }
    ],
    categories: [
      {
        categoryId: 72,
        preference: 1
      }
    ]
  }
}
```

Example 2: Global Preference Only

```
{
  success: true,
  userPreferences: {
    adChoicesString: "BYVHiWRAAAAA",
    version: 1
```

```
    timestamp: 1632756313,
    globalChoice: 1,
    participants: [],
    categories: []
  }
}
```

IFRAMEs

Ad serving code routinely runs in IFRAMEs. To support this, User Preferences API implementers must insert an empty IFRAME into the parent frame named `daaAdChoicesSupported`. This must be inserted before any ad serving code runs. Ad serving code will inspect to see if there is a frame named this in any ancestor in which case it is running inside an IFRAME.

In this case, ad serving code will use postMessage() to dispatch a message out of the IFRAME which will be handled by the User Preferences API implementer. The message has the following form:

```
{
  daaGetAdChoices: {
    id: "id string"
  }
}
```

Where “id” is an arbitrary string set by the requester to uniquely identify the request to match it up to the response.

The User Preferences API implementer will, in turn, respond to the request by using postMessage() to provide a message in this form to the requesting IFRAME:

```
{
  daaAdChoicesResponse: {
    id: "id string",
    success: [true/false], // As per the AdChoices object
    userPreferences: {} // User preferences object and subordinates
  }
}
```

In-App Details

The mobile app environment is complicated by the fact that activity spans two technologically distinct environments: the native app environment, and WebView(s) in which ads are displayed.

CMPs provide for user consent and IBA preferences management in-app by providing an SDK that the application developer includes in the code for the app. Companies that provide advertising for app publishers (i.e. ad networks, ad exchanges, mediation platforms etc.) provide an advertising SDK ("ad SDK") which is responsible for managing the ad space and providing the WebView in which the ads are displayed. In some cases, a single company may act as both CMP and advertising provider, and provide a single consolidated SDK. In that case, they take on both the roles described as "CMP SDK" and "ad SDK" in the following text.

DAA Recognized CMP SDKs must store AdChoices preferences using the operating system's mechanism for storing user settings. In the case of iOS, this is NSUserDefaults. In the case of Android, this is SharedPreferences.

DAA Recognized CMPs must store an AdChoices string, formatted per the AdChoices String specification, in a key named `daaAdChoicesString`. This key should **not** be set until the CMP has looked up stored preference information. Native code wishing to consume the information should use the OS' provided mechanism to listen for changes, i.e. NSUserDefaultsDidChangeNotification for iOS or SharedPreferences.OnSharedPreferenceChangeListener for Android.

Ad SDKs must retrieve the stored `daaAdChoicesString` user setting and listen for any subsequent changes. They then provide the JavaScript API inside any webviews used for ads, so that ad serving code running in the webview can access the information.

Typical Implementation Approach

Web

CMPs

At page load, the DAA Recognized CMP will ensure that the `daaGetAdChoices()` function is at least defined and able to be called, before any ad serving code runs. However, execution of the callback function can be deferred until further script loading.

The CMP will attempt to look up stored AdChoices preferences received from the DAAs' tools. How to persist these preferences is up to the CMP, but it will typically involve storing the choices string in a third-party cookie or server-side tied to a token (mobile advertising ID, hashed e-mail, etc). This could also be cached in a first-party cookie or localStorage to speed lookup on subsequent page loads.

The callback is called asynchronously only after the attempt to retrieve AdChoices information has occurred and the CMP has determined that it has such information (or does not). It's expected that this typically means a delay of a second or more until the callback is called while lookup occurs if it is not already locally cached.

Ad Servers, Exchanges, DSPs, DMPs, CDPs, etc.

Participants who serve ads or collect data about users, such as ad servers, exchanges, DSPs, DMPs, CDPs, etc. will call the `daaGetAdChoices()` function as a way to get real-time information about the user's AdChoices preferences.

In this case, since the preferences are tied to a token (hashed e-mail, phone number, etc.), the user may not already be known by that token to the ad serving participant or they may not know the token at time of ad serving. For this reason, using the API allows for the preferences to be retrieved in use cases where the brand or publisher has provided the ID token to the CMP to identify the appropriate user preferences string. (See Appendix 1 for a data flow diagram.)

Participants should:

1. Check if the `daaGetAdChoices()` function exists in the current frame. If it does, the page has a CMP installed that supports this API, and it is directly available for the participant to use 'getAdChoices' directly to retrieve preference information.
2. If the function is not present, it is either because the API is not supported or the participant's code is inside an IFRAME on the page. The participant's code should search ancestor frame(s) to see if there is another IFRAME named `daaAdChoicesSupported` present in an ancestor. This confirms that the postMessage() approach should be used instead.

If a CMP supporting AdChoices is not found using either of those methods, participants should proceed as they usually do whenever information is not available. If participants need to know the users' preferences before ad decisioning, they should delay ad decisioning until after the callback occurs. Participants may wish to set a reasonable timeout to account for scenarios in which the CMP does not call the callback in a timely manner.

Participants should send the AdChoices string to themselves (so that it can be passed on to other parties), but may also choose to consume the structured, parsed information in the JavaScript object for convenience in notifying themselves of preferences relevant to their ad serving decisioning.

App

CMPs

The CMP SDK will attempt to look up stored AdChoices preferences received from the DAAs' tools. It will do so by querying the CMP's servers with a token such as a hashed e-mail address. If AdChoices preferences have been expressed for the token in question, they would have been sent to the CMP at the time the user expressed those preferences using the YourAdChoices tool.

On successful retrieval of preferences, the CMP SDK will persist these to user settings on-device in the manner described in "In-App Details" above.

Native Code (App Environment)

In cases where native code is integrated into the app, AdChoices preferences are retrieved by looking up the stored user setting and listening for changes as described in "In-App Details" above.

One example of this would be an analytics SDK which wishes to incorporate the AdChoices preferences into app analytics. An ad SDK provider is another such example, with additional obligations described below.

Ad SDK Providers

Ad SDK providers are responsible for the User Preferences API implementation. They will ensure that inside of any WebView in which ads are displayed, the `daaGetAdChoices()` function is at least defined and able to be called, before any ad serving code runs. However, execution of the callback function can be deferred until further script loading.

The ad SDK will attempt to look up stored AdChoices preferences from the on-device user settings and listen for any changes that may occur in the manner described in "In-App Details" above. The ad SDK conveys this information into the WebView in whatever manner they wish, so long as the User Preferences API implementation they provide has access to it.

Ad Serving Code (JavaScript in WebViews)

Participants (typically ad servers, DSPs, etc.) whose ad serving JavaScript is loaded inside a WebView will call the `daaGetAdChoices()` JavaScript function provided by the ad SDK as a way to get real-time information about the user's AdChoices preferences. They do so in the exact same manner as described in "Web" above. However, use of the `daaGetAdChoices()` function is limited to JavaScript in the WebView. If the participant in question is also the ad SDK provider, they should follow the instructions above to retrieve the AdChoices preferences within

the app environment, and provide the JavaScript API for any “downstream” JavaScript (such as ad serving code from a third-party ad server).

For example, an ad exchange may offer an ad SDK to directly manage the display of ads in the app, but may also support integrations in which the app publisher trafficks an ad tag in their publisher ad server, with this loaded in a WebView provided by some other ad SDK. In the case where the ad exchange is also providing the ad SDK, they would follow the instructions above in “Native Code (App Environment)” and “Ad SDK Providers”. In the case where their ad tag is trafficked in the publisher ad server, they never interact with the native app environment, and will call the `daaGetAdChoices()` function (provided by the ad SDK) from their ad serving code.

Example Code

CMP Stub

This is an example of a stub implementation; per “Typical Implementation Approach” above, the CMP will ensure that the `daaGetAdChoices()` function is at least defined and able to be called, before any ad serving code runs. Implementing a stub that is loaded before any ad serving code is a suitable approach.

```
(function () {

  /**
   * AdChoices User Preferences API CMP stub implementation example.
   */

  const makeStub = () => {
    const AD_CHOICES_LOCATOR_NAME = 'daaAdChoicesSupported';
    const queue = [];
    const currentWindow = window;
    let frameLocator = currentWindow;
    let cmpFrame;

    function addFrame() {
      const doc = currentWindow.document;
      const otherCMP =
!!(currentWindow.frames[AD_CHOICES_LOCATOR_NAME]);

      if (!otherCMP) {
        if (doc.body) {
          const iframe = doc.createElement('iframe');
```

```
      iframe.style.cssText = 'display:none';
      iframe.name = AD_CHOICES_LOCATOR_NAME;
      doc.body.appendChild(iframe);
    } else {
      setTimeout(addFrame, 5);
    }
  }

  return !otherCMP;
}

function daaGetAdChoicesHandler(callback) {
  if (!callback) {
    /**
     * Shortcut to get the queue when the full CMP
     * implementation loads; it can call window.daaGetAdChoices()
     * with no arguments to get the queued arguments, before it
     * overrides the function to provide the full implementation.
     */

    return queue;
  } else {
    /**
     * Push callback into queue for full CMP implementation
     * to deal with.
     */
    queue.push(callback);
  }
}

function postMessageEventHandler(event) {
  const msgIsString = typeof event.data === 'string';
  let json = {};

  if (msgIsString) {
    try {
      /**
       * Try to parse message into JSON.
       */
      json = JSON.parse(event.data);
    } catch (ignore) {
    }
  } else {
    json = event.data;
```

```
      }

      const payload = (typeof json === 'object') ?
json.daaGetAdChoices : null;

      if (payload) {
        window.daaGetAdChoices(
          function (adChoicesObject) {
            let returnMsg = {
              daaAdChoicesResponse: {
                userPreferences: adChoicesObject.userPreferences,
                success: adChoicesObject.success,
                id: payload.id,
              },
            };

            if (event && event.source && event.source.postMessage) {
              event.source.postMessage((msgIsString) ?
JSON.stringify(returnMsg) : returnMsg, '*');
            }
          }
        );
      }
    }

    /**
     * Iterate up to the top window checking for an already-created
     * "daaAdChoicesSupported" frame at every level. If one exists
     * already then a CMP implementing this API is already loaded, and
     * provision of the stub does not proceed.
     */
    while (frameLocator) {
      try {
        if (frameLocator.frames[AD_CHOICES_LOCATOR_NAME]) {
          cmpFrame = frameLocator;
          break;
        }
      } catch (ignore) {
      }

      // If we're at the top and no cmpFrame
      if (frameLocator === currentWindow.top) {
        break;
      }
```

```
      // Move up
      frameLocator = frameLocator.parent;
    }

    if (!cmpFrame) {
      // No  "daaAdChoicesSupported" frame found - initialize IFRAME
      // and assign functions and event listeners.
      addFrame();
      currentWindow.daaGetAdChoices = daaGetAdChoicesHandler;
      currentWindow.addEventListener('message',
postMessageEventHandler, false);
    }
  };

  if (typeof module !== 'undefined') {
    module.exports = makeStub;
  } else {
    makeStub();
  }
}());
```

Querying the API

This example shows how a participant such as ad servers, exchanges, DSPs, DMPs, CDPs, etc. can interact with the API to retrieve user preferences.

```
(function() {
  // Check if CMP-provided function 'daaGetAdChoices' is present in
  // 'current' window instance
  let adChoicesFrame;

  if (typeof window.daaGetAdChoices !== "undefined") {

      console.log("CMP 'daaGetAdChoices' available in current
window!")
      const adChoicesCallback = (adChoicesObject) => {
        console.log(adChoicesObject);
      }
      window.daaGetAdChoices(adChoicesCallback)
  }
  else {
    // In case 'daaGetAdChoices' is not available in current window
    // fallback to IFRAME approach: look for defined CMP IFRAME
    // and store its reference for later use in postMessage
```

```
  // Start at 'current' window
  let frame = window;

  // Store CMP frame reference with current variable

  // map of calls
  const cmpCallbacks = {};
  while(frame) {

    try {
      if (frame.frames['daaAdChoicesSupported']) {

        adChoiceFrame = frame;
        break;

      }

    } catch(ignore) {}

    if(frame === window.top) {

      break;

    }

    frame = frame.parent;

  }
 }

/**
 * Set up a daaGetAdChoicesClient proxy method to do the postMessage
 * and map the callback. From the client view, this function behaves
 * identically to the AdChoices CMP API's `daaGetAdChoices` call
 */

 window.daaGetAdChoicesClient = function(callback) {

   if (!adChoicesFrame) {

     callback({msg: 'CMP not found'}, false);

   } else {

     const adClientId = Math.random() + '';
```

```
      const msg = {
        daaGetAdChoices: {
          id: adClientId,
        },
      };

      /**
       * Map the callback for lookup on response
       */

      cmpCallbacks[adClientId] = callback;
      adChoicesFrame.postMessage(msg, '*');

    }

  };

  function postMessageHandler(event) {

  /**
    * When we get the return message, call the mapped callback
    */

    let json = {};

    try {

      /**
        * If this isn't valid JSON then this will throw an error
        */

      json = typeof event.data === 'string' ? JSON.parse(event.data) :
event.data;

    } catch (ignore) {}

    const payload = json.daaAdChoicesResponse;

    if (payload) {

      /**
        * Messages we care about will have a payload
        */
      if (typeof cmpCallbacks[payload.id] === 'function') {
```

```
        /**
         * Call the mapped callback and then remove the reference
         */

        cmpCallbacks[payload.id](payload);
        cmpCallbacks[payload.id] = null;

      }
    }
  }

  window.addEventListener('message', postMessageHandler, false);

}());
```

Appendix 1: Data Flow Illustration – Web Scenario