HTML5APPS

DELIVERABLE D1.1

STANDARDIZATION REPORT

Project	
Grant Agreement number	317862
Project acronym:	HTML5Apps
Project title:	HTML5Apps: Closing the Gaps
Funding Scheme:	Coordination & Support Action
Date of latest version of Annex I against which the assessment will be made:	May 24, 2013
Document	
Deliverable number:	D1.1
Deliverable title	Standardization report
Contractual Date of Delivery	M12
Actual Data of Delivery:	M12
Editor(s):	Dr. Dave Raggett
Author(s):	
Reviewer(s):	
Partipant(s):	
Work package no.:	1
Work package title:	WebOS APIs
Work package leader:	Dr. Dave Raggett
Distribution:	PU
Version/Revision:	
Draft/Final:	Final
Total number of pages (including cover):	298
Keywords:	

DISCLAIMER

HTML5Apps is a project funded in part by the European Union.

TABLE OF CONTENTS

1. INTRODUCTION

This report describes the HTML5Apps project's achievements in in terms of standardizing WebOS APIs (including standardization documents).

At the outset of the HTML5Apps project, HTML5 standards were designed to cope with the user visiting untrusted web sites, necessitating a cautious approach to security that narrowly limited what a particular website can do (limited access to OS, network, and browser data through browser sandbox, avoid fingerprinting of users etc.). This limited the type of apps that could be written using HTML5.

It was assumed that closing the gap between HTML5 apps and native apps would require defining a runtime environment, security model, and associated APIs for building Web applications with comparable capabilities to native applications. This means stronger integration with the host platform than is the case for traditional web pages.

Today’s Web operating systems such as Tizen and FirefoxOS typically include the following components for which no standardized solution exists today:

- Execution Model: A description of the execution model and associated APIs for HTML5 applications, that differs from the traditional browser-based execution model.
- Security Model: A description of the security model and associated APIs for HTML5 applications that differs from the traditional browser-based security model.

Moreover, Web operating systems include a number of APIs which are also not standardized:

- Alarm API: An API to manage the system's alarm daemon.
- Contacts API: An API that enables complete management of the device's address books.
- Messaging API: An API to send and receive messages (e.g. SMS, MMS, Email, and IM) as well as manage messages stored on the device.
- Telephony API: An API to interact with the phone system, for instance to dial a number, pick up a call, route to voicemail, access the call log, etc.
- Raw Sockets API: An API to manipulate low-level connections (e.g. TCP, UDP), including the ability to listen for incoming connections.
- Bluetooth API: A low-level API to interact with the Bluetooth hardware available on some devices.
- Browser API: An API that provides all the necessary items to build a Web browser that aren't otherwise available. Most notably, this provides all that is needed in order to safely instantiate a viewport onto the open Web, pretend that such a viewport is the top level window even if the browser's chrome is itself written using Web technology, etc..

- Calendar API: An API that enables complete management of the device's calendars.
- Device Capabilities API: An API that exposes the capabilities available to the device.
- Idle API: An API to be notified when the user is idle.
- Media Storage API: An API to manage the device's storage of specific content types (e.g. pictures).
- Network Interface API: An API to manipulate network interfaces (mobile, WiFi, etc.), such as listing available networks, current strength, etc., as well as configuring and enabling them. Potential uses include offloading connections from mobile networks to WiFi, enabling high priority mobile data connections and control of other network features.
- Secure Elements API: An API enabling the discovery, introspection, and interaction with hardware tokens (Secure Elements) that offer secure services such as tamperproof storage, cryptographic operations, etc.
- System Settings API: An API to manage the system's settings (e.g. time/clock settings, and personal preferences including privacy preferences).

For HTML5 apps to realize their full potential as non-proprietary, open alternative to today’s native app environments, further functionality needs to be added to the relevant standards.

In order to develop these standards, members of the HTML5Apps project team are filling in the role of so-called “W3C team contacts” in relevant W3C standardization Working Groups. A W3C team contact acts as the interface between the Group Chair (“Chair”), Group Members, and the W3C Team. Many of the team contact’s tasks involve helping the Chair complete his or her roles, while others involve direct action from the Contact. The team contact role is largely one of communication. This involves becoming as aware as possible of the technical requirements and issues in the group, and simultaneously being aware of the general architecture of the Web as evolving in the other work of W3C. In particular, the work of team contacts include the following tasks:

- Assist Group organizers in maintaining charter and convening Group
- Monitor group participation and operations
- Monitor levels of active participation and address as needed.
- Serve as Contact between WG and rest of the W3C Team (team contacts of other groups, marketing, management etc.)

This report is structured as follows: In Section 2, we report on the status of work on the execution and security model, with particular focus on the work on permissioning. In Section 3, we report on the status of work on individual APIs. In Section 4, we give an overview of future work planning. Section 5 concludes this report. Appendices provide background materials and standardization documents.

Please note that this document includes hypertext links to background materials including draft specifications. This can be followed when viewing the electronic version of this document.

2. EXECUTION AND SECURITY MODELS

2.1. STATUS OF WORK

Work started with submissions from working group participants in the W3C Systems Applications (SysApps) WG:

Execution Model

A description of the execution model and associated APIs for system applications, particularly how the execution model differs from the traditional browser-based execution model. Example: Strawman proposal from Google.

Security Model

A description of the security model and associated APIs for system applications, particularly how the security model differs from the traditional browser-based security model. Examples and further background:

- Strawman proposal from Google,
- B2G Security Model,
- W3C Workshop in 2008 on Security for Access to Device APIs from the Web,
- The WAC core security specifications
- The BONDI App Security Framework
- Chrome extensions security model and permissions
- The webinos security model
- The Widgets security model landscape analysis from 2008
- The security controls introduced by 'Gibraltar'

These contributions were reworked by editors from Mozilla and Samsung into a draft specification.

- Runtime and Security Model for Web Applications

The draft described:

- How an application is defined through an application manifest and how it can be installed, updated and packaged.
- The eventing model to handle the different stages of the "app lifecycle" (launching, pausing, resuming, terminating).
- The offline execution model.
- Permission and security model for system APIs/capabilities that should only be granted for suitably trusted applications.

In Autumn 2013, the working group decided to split the initial proposal into several separate work items, and to proceed on them separately:

- The specification of a JSON based manifest format (edited by Intel and Mozilla. The application manifest is used to attach metadata, application icons, and can be used to describe the set of system level capabilities that the application requires. See Appendix C
- The App URI specification (edited by Mozilla) defines a URI scheme for use by HTML5 applications to obtain resources packaged as part of the application, e.g. image resources referenced by an IMG element. This specfication reached last call in May 2014. See Appendix D
- The App Lifecycle specification (edited by Intel) provides APIs for managing the lifecycle of an application and associated events. The App Lifecycle specification allows apps to respond to changes in the application lifecycle (e.g. launch and terminate events), to handle events sent by the system (e.g. push notifications), and to handle scheduled wakeup calls. See Appendix E.
- Permissioning: HTML5Apps staff created a whitepaper analyzing existing approaches and organized a special meeting on Trust and Permissions in September 2014.

2.2. TRUST AND PERMISSIONS

In order to help the Working Group move to a shared position that maximizes interoperability of trusted web applications, the HTML5Apps staff:

- created a whitepaper analyzing existing approaches (see Appendix A),
- organized a special meeting on Trust and Permissions (see Appendix B).

W3C has already taken some steps with specifications like the Geolocation API that require user consent. Others are underway, e.g. for media capture and switching to full screen mode. The question is how to proceed towards a more comprehensive approach to trust and permissions for the Open Web Platform.

2.2.1. White Paper

HTML5Apps staff started a survey on permission handling in existing Web and native application platforms as preparation for a cross working group meeting on trust and permissions in the Open Web Platform.

- Whitepaper: Handling Trust and Permissions in Web Applications, Dave Raggett, April-July 2014, see Appendix A

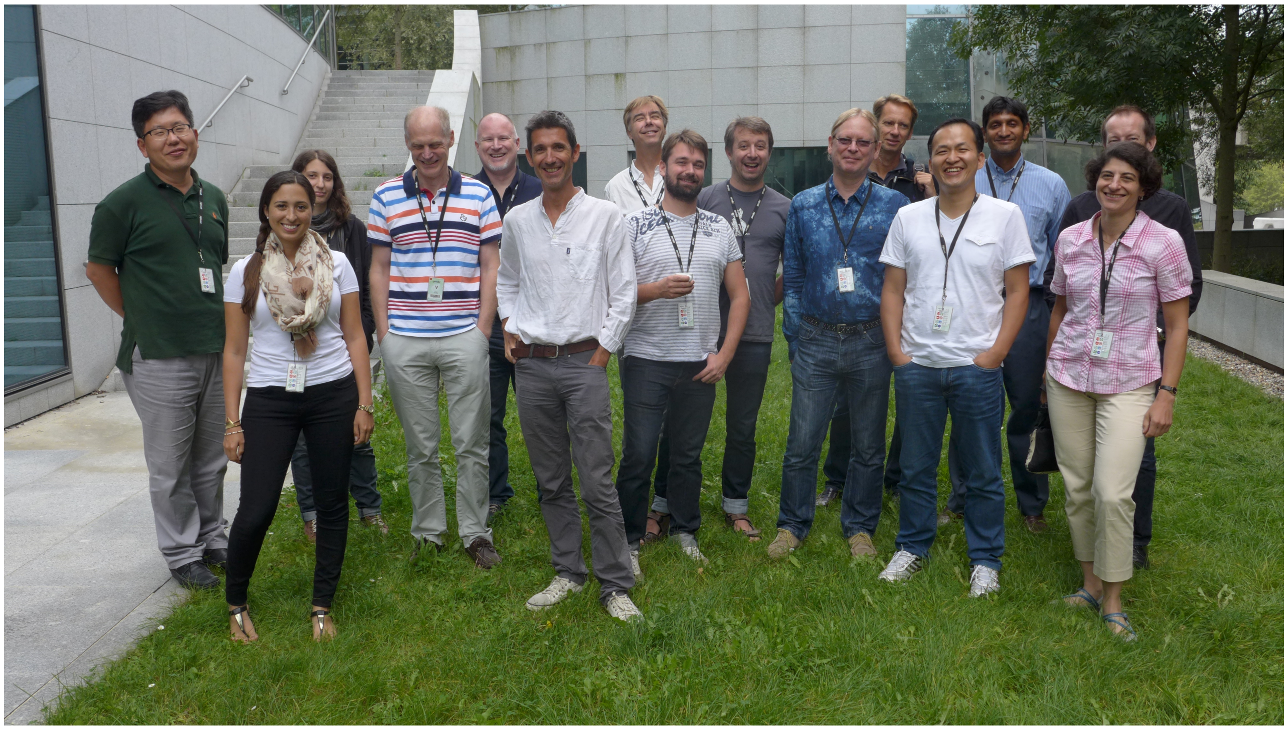
The paper looks at Google's Android, Apple's iOS, Microsoft Windows Runtime and Windows Phone, Blackberry 10 Native Apps, PhoneGap/Apache Cordova, Adobe AIR, Google Chrome Apps, Mozilla's Firefox OS, Ubuntu Web Apps, Nokia's Cloudberry, Tizen, and the W3C Open Web Platform. Further work is anticipated for automotive and TV based platforms. This paper is based upon publicly available information, and has varying levels of detail according to what information could be found. The paper also includes the email discussion on the SysApps WG list on permissions UI and necessary API. The paper concludes with some questions for further study.

2.2.2. Paris Meeting

The HTML5Apps project organized a two day meeting on trust and permissions in Paris, France on 3-4 September 2014, hosted by Gemalto. The Minutes were made available to the public (see Appendix B).

The meeting includes a broad representation of browser vendors and other organizations:

- Dave Raggett, W3C
- Dominique Hazaël-Massieux, W3C (remote)
- Robin Berjon, W3C (webapps etc.)
- Wendy Seltzer, W3C (security, privacy) 2nd day only
- Stefan Håkansson, Ericsson (Co-chair Media Capture TF, WebRTC)
- Philipp Hoschka, W3C
- Giridhar Mandyam, Qualcomm
- Claes Nilsson, Sony Mobile
- Wonsuk Lee, Samsung
- Vadim Draluk, GM (automotive)
- Adrienne Porter Felt, Google
- Jonghong (Jonathan) Jeon, ETRI
- Steven Woolcock, Apple
- John Hazen, Microsoft
- Stephanie Ouillon, Mozilla
- Kenneth Rohde Christiansen, Intel
- Olivier Potonniee, Gemalto
- Anssi Kostiainen, Intel (remote)
- Virginie Galindo, Gemalto (remote)


Credits to Johnathon Jeon.

We started with a short history of the SysApps Working Group. This group set out to standardize around 15 APIs for use in packaged and hosted apps.

We then reviewed the approaches taken to date by the Open Web Platform (OWP), iOS, Android, Windows Phone, Chrome Apps, Firefox OS and General Motor's approach for their automotive apps.

For geolocation the **current HTML5 standard** requires user consent at the time of use. By contrast, the Full Screen API asks for consent after entering into full screen mode. The user can decline, forcing the browser back to the regular window mode. The aim here is to combat phishing attacks.

Apple's **iOS** likewise takes the time of use approach. The context in which the prompts occur make it easier for users to understand what the prompts are for as compared to asking for permissions at install time. Steven Woolcock said that in some cases it would be desirable to ask for permissions prior to use. One example is where parents want to control what permissions are appropriate for apps used by their children.

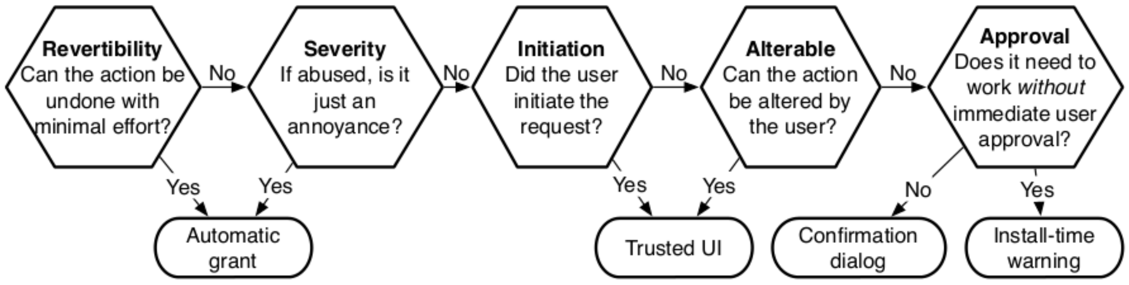
Adrienne Felt summarised how permissions are used in **Android**. There are something like 150 permissions available. The ones that developers think they will need have to be declared in the application manifest and users have to give their consent as a precondition for installing an app. A few permissions are only available to applications certified by Google. Dom gave the API for bricking a stolen phone as an example. Adrienne cited a study that showed that for the most part developers only request the permissions that their app actually needs. Users think more about what apps would do with their data, and not about the full range of possibilities that the permissions enable. It was noted that the Android permission model trains users to click through the consent dialogue in order to try out the newly installed app. In many ways users would prefer to know whether the app is trustworthy or not and not have to see the legalese of the permissions dialogue.

For **Windows Phone**, John Hazen (Microsoft) reported that Microsoft have chosen to minimize the total number of permissions available to developers and end users, ending up with around 12 coarse grained permissions. They have explicit prompts only where API would expose something about the user in real-time. Microsoft has a review process to weed out malware and can revoke apps when mis-behaving apps have been released to the wild. For enterprise apps, it is the network adminstrators who determine where apps have permission to access corporate resources.

Stéphanie Ouillon (Mozilla) presented the **Firefox OS** Permission Model which currently applies only to packaged apps. These are apps where the complete set of files have been zipped together for local installation on the phone or tablet. There are three levels: regular web apps, privileged permissions for apps signed by the Mozilla marketplace, and certified permissions which are only available to apps from Mozilla and its business partners. Firefox OS prompts users for consent at the time of use, but only for those permissions that users can be expected to understand. Claes Nilsson described work by Sony on extending the Firefox OS permissions model to hosted apps, based upon the manifest and digital signatures with a white list of trusted signatory parties.

Stefan Håkansson (Ericsson) talked about work at Ericsson on an extension of W3C Web Workers as a basis for accessing privileged APIs. The web application uses a simple JavaScript wrapper to invoke the API exposed by the trusted script of the Provider Worker via the HTML5 postMessage mechanism. One benefit is the ability to replace several lower level browser consent prompts with a single higher level and more pertinent prompt.

We then reviewed lessons from academic studies. Adrienne presented the following diagram from her dissertation (Towards Comprehensible and Effective Permission Systems)as a basis for matching the permission mechanism to the context:


Revertibility
Can the action be undone with minimal effort?
Severity
If abused, is it just an annoyance?
Initiation
Did the user initiate the request?
Alterable
Can the action be altered by the user?
Approval
Does it need to work *without* immediate user approval?
No
No
No
No
Yes
Yes
Yes
Yes
No
Yes
Automatic grant
Trusted UI
Confirmation dialog
Install-time warning

We looked at Roesner et al.'s work on trusted UI where user actions on trusted UI controls invoke privileged APIs. The browser ensures that the controls are only active when unobscured by anything else, and that app generated UI events are ignored, so that only genuine user interaction can invoke the control. Standardization of this approach requires a thorough understanding of the use cases.

In a discussion of considerations for permission handling, we agreed that listing permissions in the app manifest together with explanations on what they are needed for is useful for reviewers even if it isn't shown to end users. Moreover, if as an end user you trust the review process, then you don't need to be asked for consent for individual permissions. Trust could be on the basis of an app store's review process, a well known brand, or endorsement by trusted third parties. The review process is also relevant to privacy. End users find website privacy policies hard to understand, so this is something where trust delegation could be applied.

Whilst browser vendors are free to innovate around trust, developers will want standard ways to manage endorsements, especially for hosted web apps. An example of browser innovation, is the ability for a browser to flag suspicious apps for review based upon the pattern of APIs the app uses. This is a promising approach for detecting apps that are fingerprinting devices. Browsers could also help with embedded apps, e.g. where an app is funded through the advertisements that it embeds. Users may trust the app with personal details, but not want to disclose these to the companies providing the ads.

The meeting concluded with discussion on areas where there is good agreement, and areas where there is still some way to go to bring companies to a rough consensus. One work item is a proposal from Google for a permission testing API. This could be taken up by either the Web Apps or Device APIs Working Groups. Several of the participants were in favour of launching a W3C Community Group to work on best practices with a view to ensuring consistent approaches to API design across W3C Working Groups. This CG would review existing practices and also look at new approaches such as trusted UI. We agreed that whilst we would focus on hosted apps for the Open Web Platform, we shouldn't rule out the re-use of standards for packaged apps. Dave Raggett (W3C/HTML5Apps) took an action to organize a break-out session on trust and permissions in the OWP at the late October TPAC meeting in Santa Clara.

3. APPLICATION PROGRAMMING INTERFACES (APIS)

This section gives an overview of the standardization status of the APIs standardized within the HTML5Apps project.

3.0.1. Alarm API/Task Scheduler API

The purpose of the alarm API is to manage the system's alarm daemon.

In the reporting period, the Alarm API was renamed as the Task Scheduler API (edited by Samsung) as a more accurate name given the decision to avoid using Calendar time for scheduled tasks and the complications of dealing with time zones, especially for travellers flying across zones. It was felt that a full featured clock and alarms application could be built on top of the task scheduler API and the JavaScript Date object. See Appendix D.

3.0.1. Contacts API

An API that enables complete management of the device's address books.

In the reporting period, as discussion proceeded, it became clear that it would be beneficial to refactor the Contacts Manager API to enable the use of a data store that is shared across multiple applications. The advantage of doing so is to enable flexible data query mechanisms that can evolve separately from the underlying data store API. Christophe Dumez (Samsung) contributed a draft Contacts Manager API (edited by Samsung and Telefonica) layered on top of the Data Store API designed by Gene Lian (Mozilla).

In the reporting period, new editor's drafts were published for the Contacts Manager API (edited by Samsung and Telefonica). See Appendix E.

3.0.1. Messaging API

An API to send and receive messages (e.g. SMS, MMS, Email, and IM) as well as manage messages stored on the device.

In the reporting period, a new editor's draft of the Messaging API (edited by Intel and Telefonica) was published. See Appendix F.

3.0.1. Telephony API

An API to interact with the phone system, for instance to dial a number, pick up a call, route to voicemail, access the call log, etc.

In the reporting period, a new editor's draft of the Telephony API (edited by Mozilla, Intel, Telefonica and the University of Oxford) was published. See Appendix G.

3.0.1. Raw Socket API/TCP UDP Sockets API

An API to manipulate low-level connections (e.g. TCP, UDP), including the ability to listen for incoming connections. In the reporting period, the Raw Socket API was renamed to TCP UDP Sockets (edited by Sony Mobile) to make its scope more self evident.

In the reporting period, a new editor's draft of the TCP UDP Sockets API (edited by Sony Mobile) was published. See Appendix H.

3.0.1. Bluetooth API

A low-level API to interact with the Bluetooth hardware available on some devices.

Bluetooth provides the means for wireless access to a wide range of devices, including keyboards, pointing devices, headphones, speakers, microphones, and so forth. Bluetooth 4.0 introduced support for low energy devices with constrained communication capabilities for very long battery life.

In the reporting period, a W3C Bluetooth Community Group was set up by Google as a precursor to standardizing an API for Bluetooth Low Energy for web applications. A document describing Use Cases and a draft specification are available (both edited by Google). See Appendix I.

3.0.1. Secure elements API

An API enabling the discovery, introspection, and interaction with hardware tokens (Secure Elements) that offer secure services such as tamperproof storage, cryptographic operations, etc. In the reporting period, several drafts of a Secure Element API specification (edited by Gemalto and Deutsche Telekom) were published. See Appendix J.

4. PLANNING FUTURE WORK

In April 2014, the HTML5Apps team member serving as W3C Team Contact for SysApps initiated a survey of the companies participating in the System Applications Working Group in order to plan future work. The questions included:

1. Which SysApps work items does your company expect to implement?
2. Which SysApps work items would your like to see widely supported?
3. Do you agree that apps must have addressable HTTP/HTTPS based origins and be part of the Open Web?
4. Do you agree that web apps need access to more advanced capabilities and features than they currently have? Please provide details for use cases that are not currently addressed by the Open Web Platform.
5. Do you agree that the users of these apps must have control over the capabilities these apps have, and that users can revoke these rights?
6. Do you agree that the current permissions models are broken and we need to fix that?
7. Do you agree that a promising approach is to ask users for permission to use capabilities in the context of use? In other words, for the web run-time to ask the user when the application tries to invoke a system API, e.g. to access raw TCP/UDP sockets.
8. Do you agree with the need for upfront user consent for permissions when an app is installed or first run?
9. Do you agree that App Manifest is a one of the preconditions for apps to gain access to richer capabilities than are normally available to apps on the Open Web Platform?
10. Do you see a role for digital signatures as part of attestation for hosted apps on the Open Web Platform?
11. Please use the text box below to very briefly describe your ideas for future work items, along with your interest in implementing and deploying devices with support for the results of these work items.

Questions 1-4 and 11 address which existing or new work items have sufficient support to continue them in a new charter. The other questions were chosen to shed light on how permissions could be added to the Open Web Platform.

There was strong support for each of the following:

- web apps need access to more advanced capabilities and features than they currently have
- users should have control over the capabilities available to apps, along with the means to revoke these rights
- asking the user for permission at the time of use is promising, although not appropriate for all capabilities
- asking the user for consent up front when the app is "installed" or first run is also of value

- app manifests should be one of the preconditions for apps to gain access to richer capabilities

There was weak interest in the potential for digital signatures as part of attestation for hosted apps on the Open Web Platform. We didn't get many suggestions on ideas for future work other than for Bluetooth profiles support, and for continued work on the trust/permissions model as an extension of existing practice on the Open Web Platform.

Here are the numbers for which APIs people have plans to implement, and which APIs people would like to see widely deployed. The third number is the sum of the previous two and gives a broader feel for the level of interest:

App URI	4	5	9
TCP UDP Sockets	4	4	8
Task Scheduler	2	5	7
Bluetooth	3	4	7
Media Storage	3	4	7
Network Interface	4	3	7
App Lifecycle	3	3	6
Contacts	2	3	5
Data Store	2	2	4
Device Capabilities	2	2	4
Idle	2	2	4
Secure Elements	2	1	3
Calendar	1	1	2
System Settings	1	1	2
Messaging	1	-	1
Telephony	1	-	1

Based on these results, at the time of writing, there are ongoing discussions on prioritization of work in the future, which may lead to a somewhat different regrouping of work items into W3C standardization groups.

At the time of writing, the discussions about future work planning are still ongoing.

5. CONCLUSIONS

In the reporting period, the HTML5Apps project supported the publication of draft standardization documents serving the project's objective to “Standardize OS level APIs for HTML5 apps”.

In the area of a **runtime and security model**, progress was made on the definition of a manifest format, an App URI specification and an App lifecycle specification. There are active discussions around permissioning, supported by HTML5apps with all key players involved and interested in collaboration (Apple, Google, Microsoft and Mozilla in particular).

In the area of **APIs**, progress has been made on nine specfications: an Alarm API/Task Scheduler API, a contacts API, a messaging API, a telephony API, a raw Socket API/TCP UDP Sockets API, a Bluetooth API, a Secure elements API and a Device Capabilities API.

In the next reporting period, we expect continued progress on the runtime model, permissioning as well as on API specifications.

My thanks to my colleagues for their help in preparing this report.

APPENDICES

This contains copies of published API specifications and further background material relating to the goals of the Web OS APIs work.

A. WHITEPAPER: HANDLING TRUST AND PERMISSIONS IN WEB APPLICATIONS

This is a work in progress and incomplete, comments are welcome!

Dave Raggett, W3C
July 2014

This white paper surveys both Web and native application platforms for how they approach the challenge of addressing trust, especially for capabilities requiring elevated permissions. The motivation for this work is to prepare for discussions on a road map for shared open standards for permissions for the Open Web Platform, as developers demand richer capabilities comparable to those available on native app platforms.

A.1. INTRODUCTION

The Open Web Platform is based upon open standards and is supported on billions of devices. It is the only vendor neutral platform that spans such a wide range of devices (e.g. desktop, smart phones, tablets, TVs and cars). For developers seeking to reach a wide range of devices and operating systems, the Web is the obvious choice. However, the success of the Web has encouraged proprietary platform owners to support native applications and app stores. This has been very successful on mobile devices, where developers have been able to take advantage of vendor support for a comprehensive range of APIs. A performance and capability gap has emerged between native and Web apps. Hybrid approaches have emerged which allow developers to use standard web technologies together with proprietary extensions, and then compile to the native app platform.

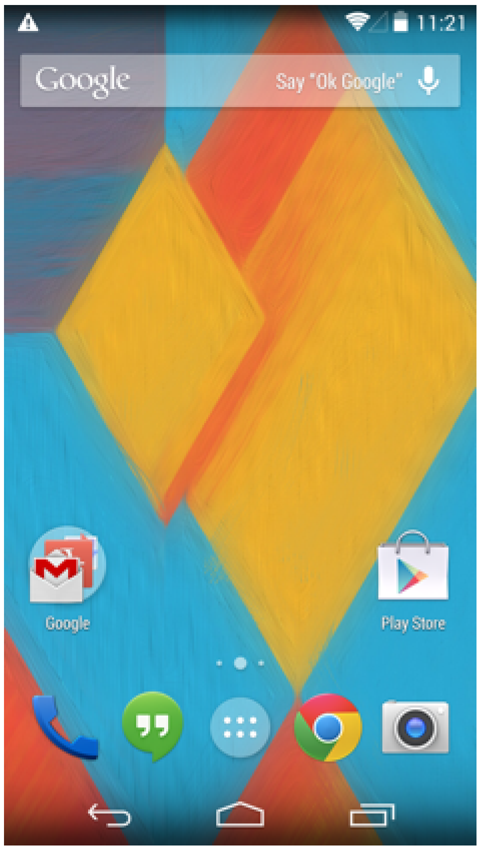
Web applications are traditionally hosted by HTTP servers, with the various resources making up the application being loaded dynamically by the web run-time (i.e. a Web Browser). Web applications can also be packaged for local installation, akin to native applications. There is a lack of interoperability for packaged apps due variations across platforms, e.g. for the associated manifests, and the use of proprietary APIs. This whitepaper surveys the field, but does not claim to be fully comprehensive. However, it should provide a broad picture of the approaches that have been taken in respect to handing trust and permissions, and potential ways to move towards a general consensus on how to extend the Open Web Platform.

Developer tools range widely in their sophistication. Some require advanced programming skills, whilst others are aimed at end users. Note that some of the platforms listed below are no longer available.

A.2. NATIVE PLATFORMS

This section looks at trust and permissions for native application platforms.

A.2.1. Google's Android Platform



Android provides an extensive suite of APIs for applications written in the Java programming language. Each application includes a manifest with a declaration of the permissions that the app needs. Users are required to give their consent before the application is installed. The user may be asked again when the manifest for app that is about to be upgraded requests an expanded set of permissions.

The Android permissions are as follows:

- ACCESS_CHECKIN_PROPERTIES - Allows read/write access to the "properties" table in the checkin database, to change values that get uploaded.
- ACCESS_COARSE_LOCATION - Allows an app to access approximate location derived from network location sources such as cell towers and Wi-Fi.
- ACCESS_FINE_LOCATION - Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
- ACCESS_LOCATION_EXTRA_COMMANDS - Allows an application to access extra location provider commands
- ACCESS_MOCK_LOCATION - Allows an application to create mock location providers for testing
- ACCESS_NETWORK_STATE - Allows applications to access information about networks
- ACCESS_SURFACE_FLINGER - Allows an application to use SurfaceFlinger's low level features.
- ACCESS_WIFI_STATE - Allows applications to access information about Wi-Fi networks
- ACCOUNT_MANAGER - Allows applications to call into AccountAuthenticators.
- ADD_VOICEMAIL -
- AUTHENTICATE_ACCOUNTS - Allows an application to add voicemails into the system.
- BATTERY_STATS - Allows an application to collect battery statistics
- BIND_ACCESSIBILITY_SERVICE - Must be required by an AccessibilityService, to ensure that only the system can bind to it.

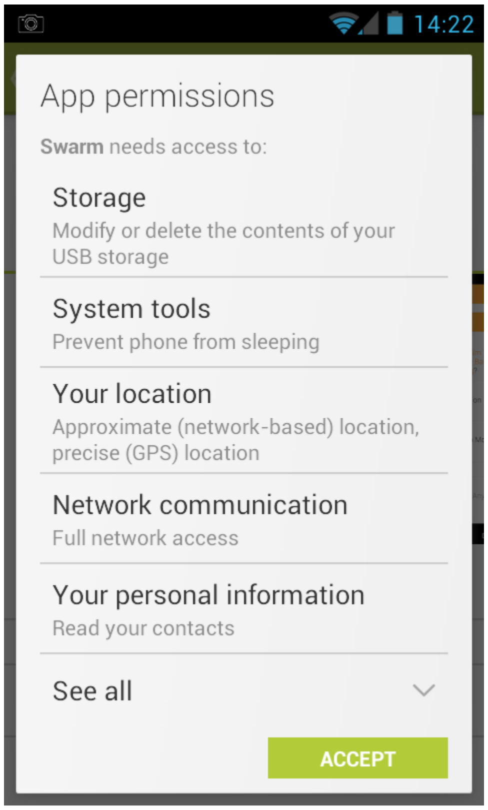
- BIND_APPWIDGET - Allows an application to tell the AppWidget service which application can access AppWidget's data.
- BIND_DEVICE_ADMIN - Must be required by device administration receiver, to ensure that only the system can interact with it.
- BIND_INPUT_METHOD - Must be required by an InputMethodService, to ensure that only the system can bind to it.
- BIND_NFC_SERVICE - Must be required by a HostApduService or OffHostApduService to ensure that only the system can bind to it.
- BIND_NOTIFICATION_LISTENER_SERVICE - Must be required by an NotificationListenerService, to ensure that only the system can bind to it.
- BIND_PRINT_SERVICE - Must be required by a PrintService, to ensure that only the system can bind to it.
- BIND_REMOTEVIEWS - Must be required by a RemoteViewsService, to ensure that only the system can bind to it.
- BIND_TEXT_SERVICE - Must be required by a TextService
- BIND_VPN_SERVICE - Must be required by a VpnService, to ensure that only the system can bind to it.
- BIND_WALLPAPER - Must be required by a WallpaperService, to ensure that only the system can bind to it.
- BLUETOOTH - Allows applications to connect to paired bluetooth devices
- BLUETOOTH_ADMIN - Allows applications to discover and pair bluetooth devices
- BLUETOOTH_PRIVILEGED - Allows applications to pair bluetooth devices without user interaction.
- BRICK - Required to be able to disable the device (very dangerous!).
- BROADCAST_PACKAGE_REMOVED - Allows an application to broadcast a notification that an application package has been removed.
- BROADCAST_SMS - Allows an application to broadcast an SMS receipt notification.
- BROADCAST_STICKY - Allows an application to broadcast sticky intents.
- BROADCAST_WAP_PUSH - Allows an application to broadcast a WAP PUSH receipt notification.
- CALL_PHONE - Allows an application to initiate a phone call without going through the Dialer user interface for the user to confirm the call being placed.
- CALL_PRIVILEGED - Allows an application to call any phone number, including emergency numbers, without going through the Dialer user interface for the user to confirm the call being placed.
- CAMERA - Required to be able to access the camera device.
- CAPTURE_AUDIO_OUTPUT - Allows an application to capture audio output.
- CAPTURE_SECURE_VIDEO_OUTPUT - Allows an application to capture secure video output.
- CAPTURE_VIDEO_OUTPUT - Allows an application to capture video output.

- CHANGE_COMPONENT_ENABLED_STATE - Allows an application to change whether an application component (other than its own) is enabled or not.
- CHANGE_CONFIGURATION - Allows an application to modify the current configuration, such as locale.
- CHANGE_NETWORK_STATE - Allows applications to change network connectivity state
- CHANGE_WIFI_MULTICAST_STATE - Allows applications to enter Wi-Fi Multicast mode
- CHANGE_WIFI_STATE - Allows applications to change Wi-Fi connectivity state
- CLEAR_APP_CACHE - Allows an application to clear the caches of all installed applications on the device.
- CLEAR_APP_USER_DATA - Allows an application to clear user data.
- CONTROL_LOCATION_UPDATES - Allows enabling/disabling location update notifications from the radio.
- DELETE_CACHE_FILES - Allows an application to delete cache files.
- DELETE_PACKAGES - Allows an application to delete packages.
- DEVICE_POWER - Allows low-level access to power management.
- DIAGNOSTIC - Allows applications to RW to diagnostic resources.
- DISABLE_KEYGUARD - Allows applications to disable the keyguard
- DUMP - Allows an application to retrieve state dump information from system services.
- EXPAND_STATUS_BAR - Allows an application to expand or collapse the status bar.
- FACTORY_TEST - Run as a manufacturer test application, running as the root user.
- FLASHLIGHT - Allows access to the flashlight
- FORCE_BACK - Allows an application to force a BACK operation on whatever is the top activity.
- GET_ACCOUNTS - Allows access to the list of accounts in the Accounts Service
- GET_PACKAGE_SIZE - Allows an application to find out the space used by any package.
- GET_TASKS - Allows an application to get information about the currently or recently running tasks.
- GET_TOP_ACTIVITY_INFO -Allows an application to retrieve private information about the current top activity, such as any assist context it can provide.
- GLOBAL_SEARCH - This permission can be used on content providers to allow the global search system to access their data.
- HARDWARE_TEST - Allows access to hardware peripherals.
- INJECT_EVENTS - Allows an application to inject user events (keys, touch, trackball) into the event stream and deliver them to ANY window.
- INSTALL_LOCATION_PROVIDER - Allows an application to install a location provider into the Location Manager.
- INSTALL_PACKAGES - Allows an application to install packages.
- INSTALL_SHORTCUT - Allows an application to install a shortcut in Launcher

- INTERNAL_SYSTEM_WINDOW - Allows an application to open windows that are for use by parts of the system user interface.
- INTERNET - Allows applications to open network sockets.
- KILL_BACKGROUND_PROCESSES - Allows an application to call killBackgroundProcesses(String).
- LOCATION_HARDWARE - Allows an application to use location features in hardware, such as the geofencing api.
- MANAGE_ACCOUNTS - Allows an application to manage the list of accounts in the AccountManager
- MANAGE_APP_TOKENS - Allows an application to manage (create, destroy, Z-order) application tokens in the window manager.
- MANAGE_DOCUMENTS - Allows an application to manage access to documents, usually as part of a document picker.
- MASTER_CLEAR - Not for use by third-party applications.
- MEDIA_CONTENT_CONTROL - Allows an application to know what content is playing and control its playback.
- MODIFY_AUDIO_SETTINGS - Allows an application to modify global audio settings
- MODIFY_PHONE_STATE - Allows modification of the telephony state - power on, mmi, etc.
- MOUNT_FORMAT_FILESYSTEMS - Allows formatting file systems for removable storage.
- MOUNT_UNMOUNT_FILESYSTEMS - Allows mounting and unmounting file systems for removable storage.
- NFC - Allows applications to perform I/O operations over NFC
- PERSISTENT_ACTIVITY -
- PROCESS_OUTGOING_CALLS - Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether.
- READ_CALENDAR - Allows an application to read the user's calendar data.
- READ_CALL_LOG - Allows an application to read the user's call log.
- READ_CONTACTS - Allows an application to read the user's contacts data.
- READ_EXTERNAL_STORAGE - Allows an application to read from external storage.
- READ_FRAME_BUFFER - Allows an application to take screen shots and more generally get access to the frame buffer data.
- READ_HISTORY_BOOKMARKS - Allows an application to read (but not write) the user's browsing history and bookmarks.
- READ_LOGS - Allows an application to read the low-level system log files.
- READ_PHONE_STATE - Allows read only access to phone state.
- READ_PROFILE - Allows an application to read the user's personal profile data.
- READ_SMS - Allows an application to read SMS messages.
- READ_SOCIAL_STREAM - Allows an application to read from the user's social stream.
- READ_SYNC_SETTINGS - Allows applications to read the sync settings

- READ_SYNC_STATS - Allows applications to read the sync stats
- READ_USER_DICTIONARY - Allows an application to read the user dictionary.
- REBOOT - Required to be able to reboot the device.
- RECEIVE_BOOT_COMPLETED - Allows an application to receive the ACTION_BOOT_COMPLETED that is broadcast after the system finishes booting.
- RECEIVE_MMS - Allows an application to monitor incoming MMS messages, to record or perform processing on them.
- RECEIVE_SMS - Allows an application to monitor incoming SMS messages, to record or perform processing on them.
- RECEIVE_WAP_PUSH - Allows an application to monitor incoming WAP push messages.
- RECORD_AUDIO - Allows an application to record audio
- REORDER_TASKS - Allows an application to change the Z-order of tasks
- SEND_RESPOND_VIA_MESSAGE - Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls.
- SEND_SMS - Allows an application to send SMS messages.
- SET_ACTIVITY_WATCHER - Allows an application to watch and control how activities are started globally in the system.
- SET_ALARM - Allows an application to broadcast an Intent to set an alarm for the user.
- SET_ALWAYS_FINISH - Allows an application to control whether activities are immediately finished when put in the background.
- SET_ANIMATION_SCALE - Modify the global animation scaling factor.
- SET_DEBUG_APP - Configure an application for debugging.
- SET_ORIENTATION - Allows low-level access to setting the orientation (actually rotation) of the screen.
- SET_POINTER_SPEED - Allows low-level access to setting the pointer speed.
- SET_PROCESS_LIMIT - Allows an application to set the maximum number of (not needed) application processes that can be running.
- SET_TIME - Allows applications to set the system time.
- SET_TIME_ZONE - Allows applications to set the system time zone
- SET_WALLPAPER - Allows applications to set the wallpaper
- SET_WALLPAPER_HINTS - Allows applications to set the wallpaper hints
- SIGNAL_PERSISTENT_PROCESSES - Allow an application to request that a signal be sent to all persistent processes.
- STATUS_BAR - Allows an application to open, close, or disable the status bar and its icons.
- SUBSCRIBED_FEEDS_READ - Allows an application to allow read access to subscribed feeds ContentProvider.
- SUBSCRIBED_FEEDS_WRITE - Allows an application to allow writes access to subscribed feeds ContentProvider.
- SYSTEM_ALERT_WINDOW - Allows an application to open windows using the type TYPE_SYSTEM_ALERT, shown on top of all other applications.

- TRANSMIT_IR - Allows using the device's IR transmitter, if available
- UNINSTALL_SHORTCUT - Allows an application to uninstall a shortcut in Launcher
- UPDATE_DEVICE_STATS - Allows an application to update device statistics.
- USE_CREDENTIALS - Allows an application to request authtokens from the AccountManager
- USE_SIP - Allows an application to use SIP service
- VIBRATE - Allows access to the vibrator
- WAKE_LOCK - Allows using PowerManager WakeLocks to keep processor from sleeping or screen from dimming
- WRITE_APN_SETTINGS - Allows applications to write the apn settings.
- WRITE_CALENDAR - Allows an application to write (but not read) the user's calendar data.
- WRITE_CALL_LOG - Allows an application to write (but not read) the user's contacts data.
- WRITE_CONTACTS - Allows an application to write (but not read) the user's contacts data.
- WRITE_EXTERNAL_STORAGE - Allows an application to write to external storage.
- WRITE_GSERVICES - Allows an application to modify the Google service map.
- WRITE_HISTORY_BOOKMARKS - Allows an application to write (but not read) the user's browsing history and bookmarks.
- WRITE_PROFILE - Allows an application to write (but not read) the user's personal profile data.
- WRITE_SECURE_SETTINGS - Allows an application to read or write the secure system settings.
- WRITE_SETTINGS - Allows an application to read or write the system settings.
- WRITE_SMS - Allows an application to write SMS messages.
- WRITE_SOCIAL_STREAM - Allows an application to write (but not read) the user's social stream data.
- WRITE_SYNC_SETTINGS - Allows applications to write the sync settings
- WRITE_USER_DICTIONARY - Allows an application to write to the user dictionary.


14:22

App permissions

Swarm needs access to:

Storage
Modify or delete the contents of your USB storage

System tools
Prevent phone from sleeping

Your location
Approximate (network-based) location, precise (GPS) location

Network communication
Full network access

Your personal information
Read your contacts

See all

ACCEPT

This list was included to illustrate the extensive range of permissions, and that many of these are specific to the design of the platform as opposed to generic capabilties. Not all devices will support all capabilities, e.g. NFC and IR.

Android's consent form provides generic descriptions of the requested permissions, but not what the given applications will do with them.

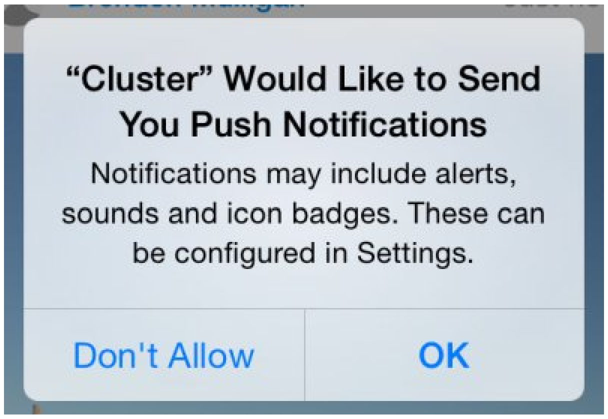
This approach encourages users to tap the ACCEPT button to proceed to try out the app, without an understanding of why the app needs these permissions. Trust is based upon the popularity and ranking of the app on the Android app store (Google Play), the name of the developer (if a well known brand), and faith in security apps like Lookout to intervene if you inadvertently are trying to install malware.

Android's approach to permissions means that developers can rely on all the permissions listed in the app's manifest as the user has to agree to them as a whole and can't pick and choose. Note that just because a permission was granted doesn't mean the device has the hardware to support the associated capability, so developers need to exercise some caution.

A.2.2. Apple's iOS Platform

Apple's iOS platform was the first major smart phone operating system, and runs apps developed with the Objective-C programming language.

- iOS Developer site
- UIApplication class reference


“Cluster” Would Like to Send You Push Notifications

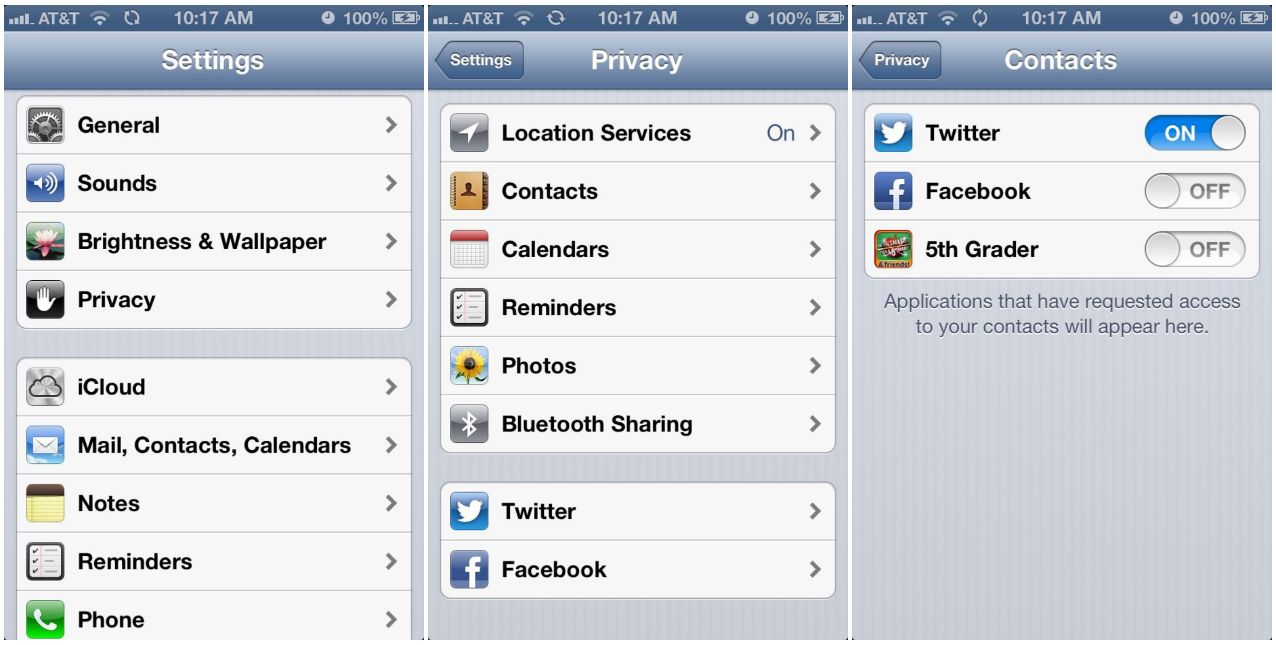
Notifications may include alerts, sounds and icon badges. These can be configured in Settings.

Don't Allow

OK

A major difference between iOS and Android is that in Android, permissions are requested up front before installing an application, whereas in iOS, permissions are requested at the time of use of a particular capability, and users may deny the request. This means that app developers need to explain how the capability will be used prior to invoking the permission dialogue. If users tap "Don't Allow", there is no easy way for them to change their mind. Brenden Mulligan's techcrunch post on the right way to ask users for iOS permissions) recommends providing clear explanations of benefits, followed by an app generated dialogue asking for a permission, and if the user says yes, this is then followed by the operating system generated permissions dialogue. If the user says no to the app generated dialog, the app can later ask again, without the user having to gone through the complicated steps to undo the Operating System's record of the user's "Don't accept" action. Note that unlike Android, developers need to write their code to fail gracefully if the permission isn't forthcoming.

Changing the permissions in iOS6*: Step 1 - activate the Settings dialogue. Step 2 - tap on "Privacy". Step 3 - tap on the category of capabilities you are interested in. Step 4 - step through the list of applications and toggle the permission on or off.*



Twitter ON
Facebook OFF
5th Grader OFF

Applications that have requested access to your contacts will appear here.

This illustrates another difference from Android. Apple has opted for a coarse set of capabilities for users to deal with when it comes to permissions, which compares to the very long list of fine grained capabilities listed above for Android.

Question*: Does iOS allow you to view and change all the permissions for a given app in a single dialogue for that app?*

A.2.3. Microsoft Windows Runtime

The Microsoft Windows Runtime enables developers to create apps with JavaScript, C#, Visual Basic, and C++ APIs for the Microsoft Windows Store. This covers devices ranging from tablets to desktops to large wall mounted touch screen displays. Having done so, you can port your app to Windows Phone for distribution on the Windows Phone Store. This wide range of devices presents challenges for designing the user experience, and Windows 8 supports a variety of navigation patterns.

- Windows App Studio is an online app creation tool for Windows and Windows Phone.

> "*Windows Runtime APIs will look and feel familiar to experienced Web developers. They represent a clean extension of standards based web development APIs*"

The Windows Runtime supports an extensive suite of asynchronous APIs, and developers can take advantage of standard components such as FileReader, Web Sockets, Geolocation, IndexedDB and others. Asynchronous programming is handled with Promises, which provide an object representing a value that has yet to be computed, or an error that has yet to occur. The Windows Runtime defines more than 800 individual classes and enums, along with a hierarchy of namespaces. The Windows Runtime APIs are split into the following categories:

- Core
- Controls
- Data and content
- Devices
- Files and folders
- Globalization
- Graphics
- Helpers
- Media
- Networking
- Printing
- Presentation
- Remote Desktop
- Security
- Social
- UI Automation
- User interaction
- Windows Preview API

Some APIs are restricted to Windows Runtime Apps and are not supported for desktop apps or browsers. APIs also distinguish between HTML and XAML as user interface markup languages. Windows Runtime apps using JavaScript are executing using the Windows Internet Explorer Standards mode. As a result some HTML and DOM APIs behave differently or aren't supported.

Anssi Kostiainen comments:

> *I believe the reason for Windows Runtime disabling some Web APIs is mainly due to security (document.write, innerHTML etc.) or UI/UX (alert, close etc.) reasons, and not due to the standards-compliant mode being used by the rendering engine. This is actually*

rather similar to how Chrome Apps [1] disable some APIs that are exposed to Chrome the browser by default.

This seems to suggest APIs should be designed in such a way they can be gated behind e.g. a promise.

A.2.4. Windows Phone

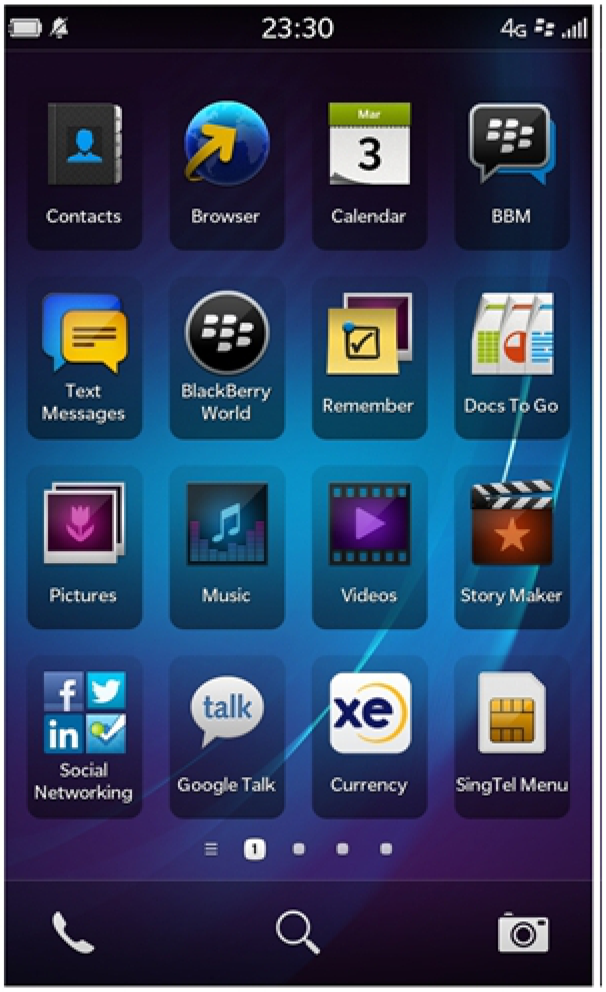
Apps for Windows Phone can be developed in JavaScript, C++, C#, and VB.NET. Windows Phone requires users to agree to a list of permissions upfront as a precondition to install apps from the Windows Phone Store. The permissions needed are listed on the left side of the app's page in the Store under the heading "App requires". Here is a list of permissions that apps can request according to the Windows Phone Central website.

- Appointments – Allows an app to access the calendar and appointment info on your phone.
- Camera – Allows an app to access the phone's built-in camera.
- Compass – Allows an app to access the phone's built-in compass, if available.
- Contacts – Allows an app to access the contact info on your phone.
- Data services – Your phone's cellular data or Wi-Fi connection.
- Gyroscope – Allows an app to access the phone's built-in gyroscope, if available.
- Location services – The approximate location coordinates of your phone.
- Photo, music, and video libraries – Allows an app to access all photos, music, and videos on your phone.
- Microphone – Allows an app to record audio from the phone's built-in microphone and to use Speech features.
- Movement and directional sensor – Allows an app to access the phone's motion sensor.

- Proximity – Allows access to the phone's Bluetooth, Wi-Fi, and near field communication (NFC) capabilities.
- Owner identity – An anonymous identifier that allows an app to distinguish one person from another, but provides no personal info.
- Phone identity – A unique device identifier that allows an app to distinguish one phone from another.
- Push notification service – Notifications that an app automatically sends to your phone in the background.
- Ringtones – Allows an app to access the ringtone collection on your phone.
- SD card – Allows an app limited access to the SD card.
- Speech recognition – Allows an app to access Speech features.
- Wallet – Allows an app to access items in your Wallet or to make payments using your Wallet.
- Web browser – Allows an app to access your phone's web browser.
- Xbox – Allows an app to access the Xbox service or your account info.

The required permissions are declared in the app manifest which is an XML file generated by Visual Studio from the app's project settings. Windows Phone is similar to Android in requiring upfront permissions.

A.2.5. Blackberry 10 Native Apps



Blackberry 10 is basd on the QNX operating system. The Blackberry 10 native SDK allows you to develop apps in C/C++ or QML (a JavaScript like scripting language for Qt). The operating system generates a dialog box automatically to request permissions from the user. The user can decide which requested permissions to grant and the choices are recorded for use when the app runs the next time, or even after the app upgrades or updates. Developers must list the permissions they want in the *bar-descriptor* xml file. Permissions are divided into categories:

- **User-granted permissions** - the user is prompted to grant or deny these permissions when the app is first run.
- **Restricted Permissions** - use of these permissions require your app to be signed in accordance with the conditions imposed by the Blackberry World storefront.
- **Developer-driven permissions** - these must be listed in the bar-descriptor file, and enable platform features that are useful in specific circumstances.

Permissions can structured into main and nested permissions. Users can choose whether to grant all sub-permissions under the main permission or just a subset.

A.3. CROSS PLATFORM FRAMEWORKS

These frameworks allow developers to create app using cross platform technologies, and either compile them into native apps that can run on a variety of operating systems, or provide a native run-time that can execute cross platform code. This can cut the time to deliver to multiple target platforms. Cross platform frameworks inherit the trust/permissions model from the native platform.

According to Research Guidance (see below), the most popular tools are PhoneGap and jQuery Mobile, followed by Adobe Air, Qt Creator, Unity 3D, Titanium, Marmalade, Sencha Touch, Xamarin, Unity Mobile, and Corona SDK. Cross platforms tools are mainly used to develop apps for games, followed by utilities, business, education and entertainment in decreasing order of popularity.

For more in-depth reviews of cross platform frameworks, see:

- Cross Platform App Development Tool Benchmarking 2013 , October 2013, Research Guidance
- In-depth profiles for major cross-platform developer tool vendors, February 2012, VisionMobile.
- Pros and Cons of the Top 5 Cross-Platform Tools, November 2013, DeveloperEconomics

The following covers just a few cross platform frameworks. A wider set is covered in the appendices. These include: Telerik, Appcelerator Titanium, Xamarin/Mono, Qt, Unity 3D/Mobile, Corona SDK, Marmalade, GINGEE, Codename One, DragonRad, RunRev LiveCode, IBM Worklight, MoSynch, RhoMobile, and Whoop. Further research is needed to identify how most of these solutions approach the challenge of permissions.

A.3.1. PhoneGap/Apache Cordova

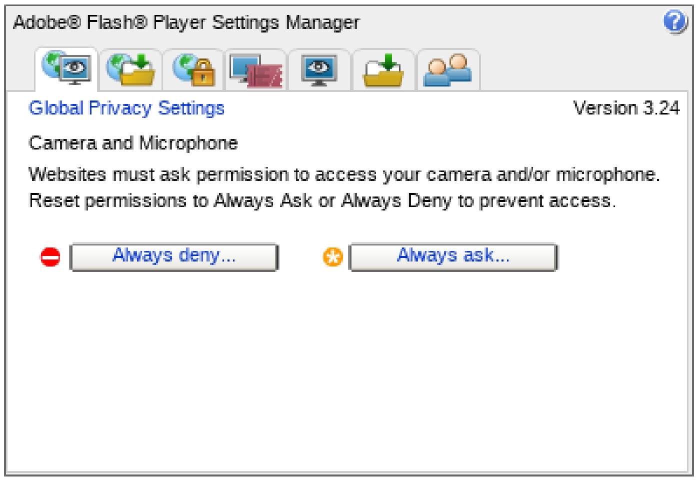
PhoneGap is an open source distribution of Cordova, which is a mobile app development framework supported by the Apache Cordova project, see the PhoneGap FAQ. It allows developers to use HTML5 and JavaScript to create native apps for Apple iOS, Blackberry, Google Android, LG webOS, Microsoft Windows Phone, Nokia Symbian OS, Tizen, Bada, Firefox OS and Ubuntu Touch. There are separate SDK's for each target platform. Cordova supports a small set of APIs, and these can be supplemented through plugins. The core plugins include:

- Battery Status - Monitor the status of the device's battery.
- Camera - Capture a photo using the device's camera.
- Contacts - Work with the devices contact database.
- Device - Gather device specific information.
- Device Motion - Tap into the device's motion sensor.
- Device Orientation - Obtain the direction that the device is pointing.
- Dialogs - Visual device notifications.
- File System - Hook into native file system through JavaScript.
- File Transfer - Hook into native file system through JavaScript.
- Geolocation - Make your application location aware.
- Globalization - Enable representation of objects specific to a locale.
- In App Browser - Launch URLs in another in-app browser instance.
- Media - Record and play back audio files.
- Media Capture - Capture media files using device's media capture applications.
- Network Information - Quickly check the network state, and cellular network information.
- Splash Screen - Show and hide the applications splash screen.
- Vibration - An API to vibrate the device.

Some of the commonly downloaded plugins include: device, console, file, inappbrowser, network-information, dialogs, splashscreen, camera and geolocation. At the time of writing this whitepaper, the cordova website listed 234 plugins.

A.3.2. Adobe AIR

AIR (Adobe Integrated Runtime) is a cross platform run-time system for desktop and mobile, based upon Adobe's ActionScript and the Adobe Flash Player, together with extensions for device capabilities such as access to the local file system, taskbar/dock integration, accelerometer and GPS. The permissions model is inherited from Flash and tailored for each target platform, e.g. iOS, Android, Blackberry, and so forth. The screen shot is for the Flash Settings panel on Linux and has tabs for:


Adobe® Flash® Player Settings Manager

Global Privacy Settings

Version 3.24

Camera and Microphone

Websites must ask permission to access your camera and/or microphone. Reset permissions to Always Ask or Always Deny to prevent access.

- Camera and microphone
- Global storage settings
- Global security settings
- Protected content playback
- Website privacy settings
- Website storage settings
- Peer assisted networking

A.4. WEB-BASED PLATFORMS

The Web is characterised by the availability of browsers on many different devices and operating systems, and from a variety of different vendors. There is good interoperability for core features, although application developers do need to consider varations in support, especially from older browsers, or for new features where the standards are still emerging. Popular web libraries like jQuery can simplify development through APIs that mask differences across browsers.

A recent trend is the emergence of platforms that combine the core features of the web with proprietary features for an alternative to native application platforms. These support server based applications in the same way as conventional web browsers, and also support installed applications, where the various components have been packaged into a single file for easy installation and offline operation.

More recently, there is growing interest in enabling server-hosted applications to work better offline using Service Workers. Related work on application manifests provides a means for developers to put metadata associated with a web application. As these mature, they are expected to provide cross vendor alternatives to packaged apps, which today need to be tailored to each platform due to variations in packaging formats across vendors.

A.4.1. Chrome Apps

Google Chrome Apps are essentially web applications that run on the Google Chrome web run-time and execute without the regular browser UI (aka *chrome*). Google supports both server hosted and packaged apps.

> *Chrome Apps deliver an experience as capable as a native app, but as safe as a web page. Just like web apps, Chrome Apps are written in HTML5, JavaScript, and CSS. But Chrome Apps look and behave like native apps, and they have native-like capabilities that are much more powerful than those available to web apps. Chrome Apps have access to Chrome APIs and services not available to traditional web sites.*

The app lifecycle has the following steps:

Installation
: The user picks the app from the app store and chooses to install it, and in the process explicitly grants the permissions the app requests in its manifest.

Start up
: The app launches with an "event page" and one or more "app pages".

Termination
: The user or the operating system can terminate apps at any time. Apps can save their state for subsequent invocations.

Update
: Apps can be updated at any time, but this doesn't effect apps while they are currently running.

Uninstallation
: Users can uninstall apps. Google Chrome ensures that all executing code and private data associated with the app are purged.

Chrome apps use the same security model as the Open Web Platform. This includes the Same Origin model, support for Content Security Policies, app local storage, and isolation between different windows for the same app. The permissions model requires upfront consent by the user. It is unclear what requirements there are for app developers to explain to the user just how each of the requested permissions will be used by the app. This makes Chrome

Apps along with Android subject to the click through effect, where users feel encouraged to give consent in order to start using the app.

The following lists the currently available permissions:

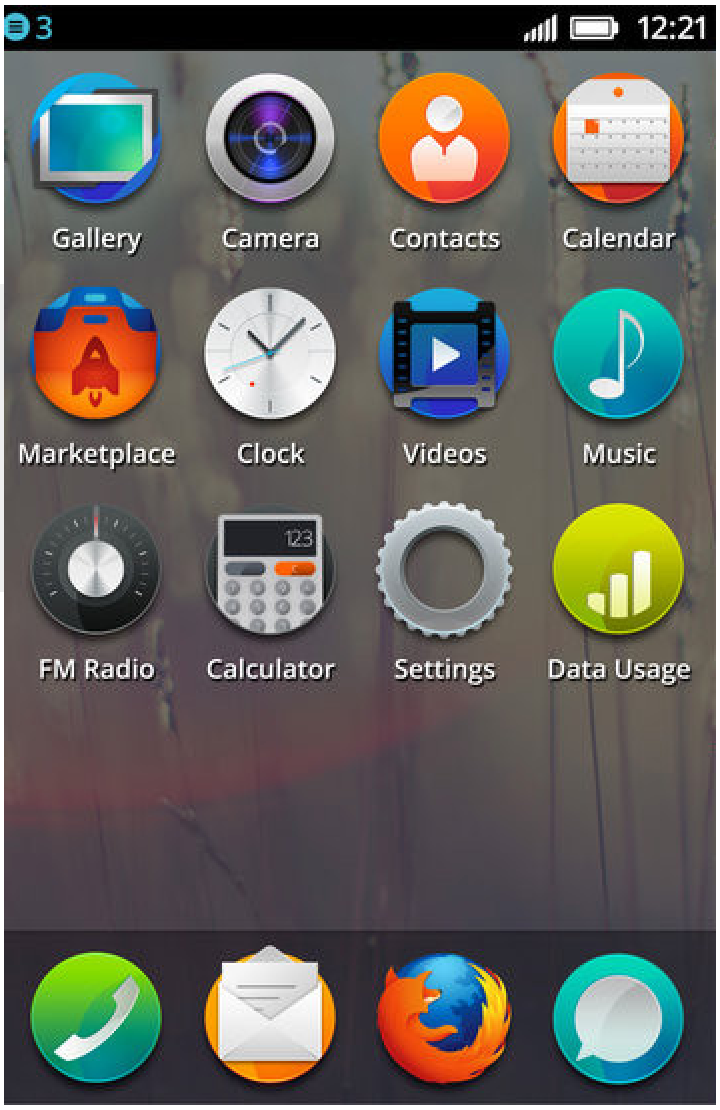
- **alarms**
- **audio**
- **audiocapture**
- **browser**
- **clipboardRead**
- **clipboardWrite**
- **contextMenus**
- **copresence**
- **desktopCapture**
- **diagnostics**
- **dns**
- **experimental**
- **fileBrowserHandler**
- **fileSystem**
- **fileSystemProvider**
- **gcm**
- **geolocation**
- **hid**
- **identity**
- **idle**
- **infobars**
- **location**
- **mediaGalleries**
- **nativeMessaging**
- **notificationProvider**
- **notifications**
- **pointerLock**
- **power**
- **pushMessaging**
- **serial**
- **signedInDevices**
- **socket**
- **storage**
- **syncFileSystem**
- **system.cpu**
- **system.display**
- **system.memory**
- **system.network**
- **system.storage**
- **tts**
- **unlimitedStorage**
- **usb**
- **videoCapture**
- **wallpaper**
- **webview**

This can be compared with the Android permissions as described above. Android provides a more extensive set of permissions that reflect the richer integration with operating system level capabilities.

Some web features aren't available for use by Chrome apps or else are supported in a different way. Google justify this as avoiding security issues and improving programming practices. Some examples include cookies and document.write. For more details see Google's page on Disabled Web Features.

A.4.2. Firefox OS

Firefox OS is a Mozilla platform for smart phones and tablets based upon a web run-time layered on top of the Linux kernel. This allows developers to create apps using HTML5, JavaScript and CSS.



> *"The webapps platform that we use in FirefoxOS and Firefox Desktop allows any website to be an app store", Jonas Sicking, 2 June 2014, Mozilla*

The way Firefox OS handles app permissions distinguishes between hosted apps and packaged apps. Hosted apps are dynamically downloaded from websites. Packaged apps are installed on the device analogous to native apps on other platforms, and are divided up into three categories:

- Web apps that don't make use of privileged or certified APIs, and may be self-published outside of the Firefox Marketplace (these are equivalent to hosted apps)
- Privileged apps that make use of privileged APIs and must be distributed through the Firefox Marketplace
- Certified apps that are able to access privileged and certified APIs, are preinstalled, and not available through the Firefox Marketplace

Privileged and certified apps are required to have content security policies. Firefox OS makes use of JSON manifest files that are linked from the HTML for hosted apps or included as part of packaged apps. All apps are required to invoke an installation method to register the manifest. This directs Firefox OS to validate the app and ask the user for approval to install the app.

Here is a list of permissions with the minimum app type required, and whether the permission is enabled by default, or results in a prompt at the time of use. Note that permissions for certified apps are intended for system level applications.

Permission	Description	Minimum	Default
alarms	schedule notification or app to be started	hosted	allow
audio capture	get audio stream from e.g. microphone	hosted	prompt
audio channel alarm	alarms from clock or calendar	privileged	allow
audo channel content	music, video	hosted	allow
audio channel normal	UI sounds, web content, music, radio	hosted	allow
audio channel notification	new email, incoming SMS	privileged	allow
browser	enables browser in iframe	privileged	allow
camera	take photos, video, record audio, control camera	privileged	prompt
contacts	read/write access contacts on device or SIM	privileged	prompt
desktop notification	display notification on desktop	hosted	prompt for hosted apps, otherwise allow
device storage music	read/write access to music stored on device	privileged	prompt
device storage pictures	read/write access to pictures stored on device	privileged	prompt

Permission	Description	Minimum	Default
device storage sdcard	read/write access to files stored on SD card	privileged	prompt
device storage videos	read/write access to video stored on device	privileged	prompt
fmradio	control fm radio	hosted	allow
geolocation	access device location	hosted	prompt
keyboard	allow app to act as virtual keyboard	privileged	allow
mobile network	access network info e.g. MCC, MNC	privileged	allow
push	enable app to wake up for notification	hosted	allow
storage	utilize appcache, indexedDB	hosted	allow
system XHR	enable cross origin XHR without CORS	privileged	allow
tcp socket	create and use TCP sockets	privileged	allow
video capture	obtain video stream from e.g. camera	hosted	prompt
attention	allow apps to open window in front of other apps	certified	allowed
audio channel ringer	incoming phone calls	certified	allowed
audio channel telephony	telephone and VoIP calls	certified	allowed
audio channel notification	forced camera shutter sounds	certified	allowed

Permission	Description	Minimum	Default
background sensors	listen to proximity events in background	certified	allowed
background service	allow apps to run in background	certified	allowed
bluetooth	low level access to Bluetooth hardware	certified	allowed
cell broadcast	fire event e.g. on emergency network notification	certified	allowed
device storage apps	read/write files in apps storage location	certified	allowed
embed apps	allow embedding of apps in mozApp frames	certified	allowed
idle	notify when user is idle	certified	allowed
mobile connection	access to info about voice and data connection	certified	allowed
network events	monitor network uploads and downloads	certified	allowed
network stats manage	access stats of data usage per interface	certified	allowed
open remote window	window.open as new process	certified	allowed
permissions	allow app to manage permissions of other apps	certified	allowed
power	turn screen on/off, control CPU, listen to lock events	certified	allowed
settings	configure and read device settings	certified	allowed

Permission	Description	Minimum	Default
sms	send and receive SMS	certified	allowed
telephony	enable telephony APIs to make and receive calls	certified	allowed
time	set current time	certified	allowed
voicemail	access voicemail	certified	allowed
webapps manage	manage installed open web apps	certified	allowed
wifi manage	enumerate networks, access strength, connect to network	certified	allowed
wappush	receive WAP push messages	certified	allowed

The Firefox OS approach to permissions implicitly grants some permissions depending upon the application type (hosted, privileged or certified) and asks the user for approval for other permissions, e.g. access to the camera. When Firefox OS doesn't prompt, trust is based upon the review performed by a human being (the App Store reviewer) who approves adding the app to the app store.

An open question is whether Firefox OS allows you to view all the permissions for a given app and choose whether to allow or deny them. It is likely that you can't deny permissions for certified apps.

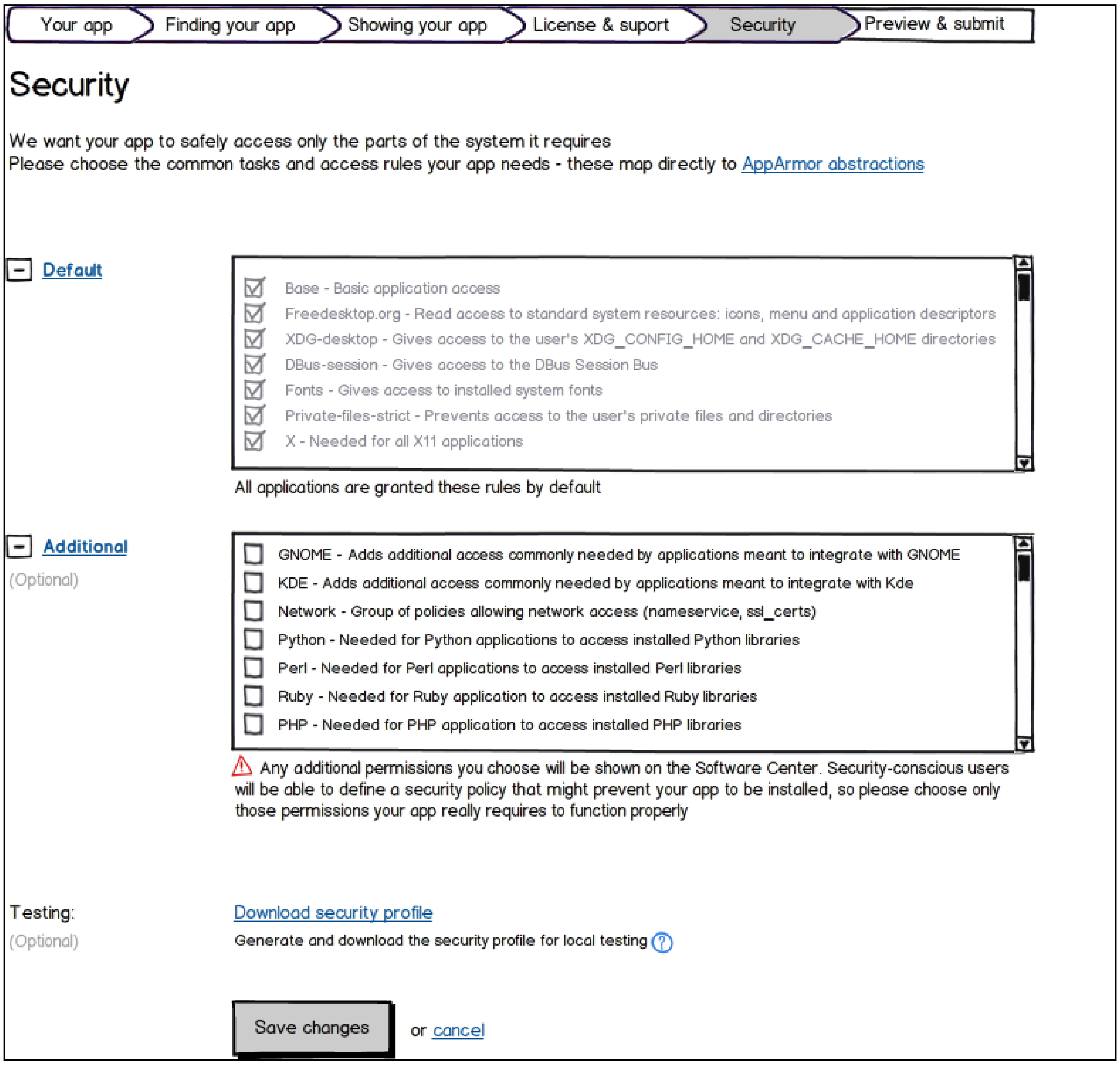
A.4.3. Ubuntu Web Apps

Note: Ubuntu also supports QML based apps, see e.g. this tutorial.

Ubuntu Web Apps enable Ubuntu users to run online applications like Facebook, Twitter, Last.FM, Ebay and GMail direct from the desktop, and treats web apps as first class citizens. This means that you can search for and invoke web apps in just the same way as for native apps. Web apps can also be selected for particular roles e.g. chat or photo sharing. Apps can access standard Web APIs as well as Ubuntu platform APIs like Content Hub, Alarms, and Online accounts, and others, such as Cordova, which provides access to system and device level functionality like camera and accelerometer, see the Ubuntu HTML5 apps developer page.

- Device and Sensors
  - org.apache.cordova.battery-status
  - org.apache.cordova.camera

    - org.apache.cordova.device
    - org.apache.cordova.device-motion
    - org.apache.cordova.media-capture
    - org.apache.cordova.vibration
- Graphical Interface
    - UbuntuUI
    - org.apache.cordova.dialogs
    - org.apache.cordova.dialogs
    - org.apache.cordova.splashscreen
- Platform Services
    - AlarmApi
    - ContentHub
    - OnlineAccounts
    - RuntimeApi
    - org.apache.cordova.inappbrowser
    - org.apache.cordova.network-information
- Multimedia
    - org.apache.cordova.media
- Language Types
    - org.apache.cordova.globalization



Security

We want your app to safely access only the parts of the system it requires
Please choose the common tasks and access rules your app needs - these map directly to AppArmor abstractions

Default

- [✓] Base - Basic application access
- [✓] Freedesktop.org - Read access to standard system resources: icons, menu and application descriptors
- [✓] XDG-desktop - Gives access to the user's XDG_CONFIG_HOME and XDG_CACHE_HOME directories
- [✓] DBus-session - Gives access to the DBus Session Bus
- [✓] Fonts - Gives access to installed system fonts
- [✓] Private-files-strict - Prevents access to the user's private files and directories
- [✓] X - Needed for all X11 applications

All applications are granted these rules by default

Additional
(Optional)

- [ ] GNOME - Adds additional access commonly needed by applications meant to integrate with GNOME
- [ ] KDE - Adds additional access commonly needed by applications meant to integrate with Kde
- [ ] Network - Group of policies allowing network access (nameservice, ssl_certs)
- [ ] Python - Needed for Python applications to access installed Python libraries
- [ ] Perl - Needed for Perl applications to access installed Perl libraries
- [ ] Ruby - Needed for Ruby application to access installed Ruby libraries
- [ ] PHP - Needed for PHP application to access installed PHP libraries

⚠ Any additional permissions you choose will be shown on the Software Center. Security-conscious users will be able to define a security policy that might prevent your app to be installed, so please choose only those permissions your app really requires to function properly

Testing:
(Optional)

Download security profile
Generate and download the security profile for local testing ?

Save changes or cancel

HTML5 apps are executed in a security sandbox (AppArmor). Each app needs to provide a security profile using a web form to generate an AppArmor policy file before upload to the Ubuntu Software Center. Users can set their own security policy, and where this conflicts with app policies, this will block the apps from being installed or executed. AppArmor supports the following restrictions:

- Limit read and write access to a pre-defined list of files and directories
- Restrict access to the network
- Limit which network destinations an app can access
- Limit what DBus services the application can call
- Prevent apps from listening in on other apps' DBus communication
- Limit what external programs an app is able to call
- Limit what signals an app can use and where it can send them
- Force external programs to run under the same restrictions as the calling app

- Display manager/server (requires kernel and userspace support from AppArmor as well as support from the display manager/server)
    - Prevent an app from listening in on another app's keyboard/ mouse input and output
    - Restrict an application's ability to perform screenshots outside of its window
    - Restrict an application's ability to access the clipboard
    - Restrict an application's ability to perform drag and drop
    - Limit/Support 3D in some manner

In addition to AppArmor, sandboxing will be required by other parts of Ubuntu, e.g.

- DConf/GSettings
    - Prevent an app from writing to system-wide or session-wide settings
    - Prevent an app from reading or writing to another app's settings
    - Prevent Gtk from loading and executing additional modules based on an app's settings
- GNOME Keyring
    - Prevent unauthorized apps from obtaining credentials
- Ubuntu Online Accounts
    - Prevent unauthorized apps from obtaining credentials
- Location service

In summary, Ubuntu web apps are subject to security policies set by the developer or the user for access to privileged APIs that extend the Open Web Platform.

A.4.4. Nokia's Cloudberry

> *"A cloud phone is a mobile device in which all customer-facing functionality is downloaded and cached dynamically from the Web, including all the applications and even the entire top-level user interface (UI) of the device."*

Cloudberry is an HTML5-based cloud phone software platform developed by Nokia Research Center. In Cloudberry, all mobile device applications are written as Web applications, including core ones such as the phone dialer, contacts, calendar, messaging, music player, and maps. Offline support relies on standard HTML5 features. Device APIs are based on official W3C Device APIs wherever applicable, and proprietary APIs are used in those areas that standards don't yet cover. The security model is based upon Web Domains, i.e. the standard security model for HTML5. Cloudberry has a permission-based security model that restricts the use of device-specific functionality (such as device APIs) to only those applications from trusted domains.

The above draws upon material from a post by Antero Taivalsaari and Kari Systa in February 2013.

A.4.5. Tizen

Tizen supports web applications as signed web widgets installed from the Tizen app store, with the standard HTML5 APIs plus Tizen specific APIs. The Tizen specific APIs are scoped to the tizen object, e.g.

```
try {
  var adapter = tizen.nfc.getDefaultAdapter() ;
} catch (err) {
  console.log (err.name +": " + err.message);
}
```

The Tizen APIs fall into the following categories:

- **Application**
    - Alarm - support for setting and unsetting alarms.
    - Application - information about running and installed applications, and controlling them.
    - Data Control - access to specific data exported by other applications.
    - Package - install or uninstall packages, and retrieve information about installed packages.
- **Communication**
    - Bluetooth - enables control over Bluetooth.
    - Messaging - allows SMS, MMS, and Email message sending and receiving.
    - Network Bearer Selection - enable users to set network bearer for a specific IP address.
    - NFC - access to NFC device(s).
    - Push - functionality for receiving push notifications.
    - Secure Element - access to Secure Elements.
- **Content**
    - Content - discover multimedia content (such as images, videos or music).
    - Download - support for downloading remote objects by HTTP request
- **Input/Output**
    - Filesystem - read/write access to device file system
    - Message Port - communication with other applications
- **Social**
    - Bookmark - access to Bookmarks.
    - Calendar - management of calendar information.
    - Call History - access to call history for cellular and VoIP calls.
    - Contact - management of contact information.
    - Data Synchronization - synchronize device data to the server using the OMA DS 1.2 protocol.
- **System**
    - Power - control power resources

    - System Information - information about the device's display, network, storage and other capabilities.
    - System Setting - system setting functionality.
    - Time - information about date, time and time zones.
    - Web Setting - manages the setting states of the web view in web applications.
- **User Interface**
    - Notification - a way to notify the user of events that happen in the application.

Widgets require authorization to access restricted APIs. The widget manifest file lists the features that the applications wants to be able to access. The manifest is represented in XML and each feature is assigned a URL based name, e.g.

```
<widget xmlns="http://www.w3.org/ns/widgets">
  <feature name = "http://example.com/api/contact" required = "false"/>
</widget>
```

Following the W3C Widget Access Request Policy (WARP), the app manifest is also used to declare which network resources (such as XMLHttpRequest, iframe, img, script, etc.) the widget would like to access, as by default, widgets are not allowed to access the network, e.g.

```
<access origin="http://example.org:8080" subdomains="false"/>
```

The Tizen web runtime grants access to features according to the policy, which sets which prompt type is to be used to request user approval.

- **blanket prompt** - User is prompted for confirmation the first time the API function is called by the widget, but once confirmed, prompting is never again required
- **session propmpt** - User is prompted once per session
- **one-shot prompt** - User must be prompted each time the restricted API is invoked
- **permit** - Use of the device capability is always permitted, without asking the user
- **deny** - Use of the device capability is always denied

Here is a sample Tizen policy file:

```
<policy-set id="Tizen-Policy" combine="first-matching-target”>
  <policy id="Tizen-Policy-Trusted" description="Tizen's policy for trusted domain"
    <rule effect=”prompt-session">  <!— rules for specific resources -->
      <condition combine="and">
        <condition combine="or">
          <resource-match attr="device-cap" func="equal" match="XMLHttpRequest" />
          <resource-match attr="device-cap" func="equal” match="externalNetworkAcces
          <resource-match attr="device-cap" func="equal" match="messaging.send" />
        </condition>
        <environment-match attr="roaming" match="true" />
      </condition>
    </rule>
    <rule effect="permit" />  <!– all other matches -->
  </policy>
</policy-set>
```

Tizen defines long lists of URLs for features, privileges, runtime, setting, and system. See also Setting Widget Configuration, which provides links to widget properties including license information, UI preferences, features, privileges, network policies, localization, and other properties.

Tizen runtime and system URLs are enums used by certain Tizen APIs such as System Information, and as such, not relevant to this paper. Settings are proprietary config.xml extensions, some of which are now being standardized in the W3C Manifest spec (e.g. orientation).

A.4.6. QNX automotive web apps

to be added

A.4.7. GM automotive web apps

to be added

GM provides an HTML5 platform for automotive apps

A.4.8. Ford SYNC AppLink

Ford SYNC is an integrated system for Ford cars with support for telephone calls, music, traffic and navigation, etc. The system is based upon Microsoft's Windows Automotive Embedded platform. SYNC AppLink is an API for apps running on iOS, Android and Blackberry mobile devices to integrate with the car's stereo system, dashboard buttons and display, via a Bluetooth or USB connection.

For hands free operation, users can control apps with spoken commands, along with speech synthesis for feedback. Ford limits access to the AppLink API to certified applications as a safety measure. Apps are available for news and information, music and entertainment, and navigation and location. It is unclear whether Ford supports HTML5 apps with AppLink. News reports indicate that Ford will switch to QNX for its next generation Sync system.

A.4.9. TV web apps (HbbTV)

to be added

A.5. W3C AND THE OPEN WEB PLATFORM

Describe the standard security framework for web apps, CORS and CSP. Then describe the approach taken in recent W3C work (DAP, Geoloc, WebRTC, Automotive). Summarise sysapps thread on including justfication in browser generated consent dialog.

The Open Web Platform (OWP) is defined by the set of standards for web protocols (HTTP, Web Sockets), HTML, CSS, JavaScript, and graphics (e.g. JPEG, PNG, SVG), and covers the core features that are widely interoperable across Web browsers and Web run-times. The security model is based upon the same origin policy, which constrains web page scripts to only accessing the execution context for pages originating from the same origin (a combination of URI scheme, host name and port number). Scripts can only access HTTP or Web Socket connections on the same origin as the page that loaded the script.

There are work arounds, e.g. dynamically adding script elements to the web page as a means for remote procedure invocation. Cross-Origin Resource Sharing (CORS) is based upon additional headers in HTTP responses that indicate which origins may request this URI. The browser/web run-time interprets these headers to relax the same origin policy. The document.domain property provides another solution for documents with a common subdomain, e.g. foo.example.com and bar.example.com.

Cross document messaging is possible with postMessage even when the documents are from different domains. One document calls postMessage to deliver data which the other document can handle by adding a listener for the 'message' event.

Content Security Policies can be set by the web page to disable potentially harmful features. This can help defend against malicious changes to scripts, e.g. those loaded from other sites, or through content injection attacks on the web page itself.

A.5.1. Permissions in the Open Web Platform

The OWP has so far dealt with permissions individually, specification by specification. A key consideration has been to minimize disclosure of personal information without the consent of the user. APIs with minimal impact on privacy are enabled for any origin. For APIs with strong impacts on privacy, the user's action to invoke a feature may be taken as implicitly granting permission for use of an API, or the browser may ask the user for explicit consent. Browsers vary in how they support re-use of a decision in further sessions, and how users can revoke such decisions.

The geolocation API is exposed by the *navigator.geolocation* object. Scripts can access the device location by calling the *getCurrentPosition()* method, passing a function to be called with the current position. The browser then asks the user for permission for the app to access the location, before invoking that function. The W3C specification requires apps to disclose the purpose for the collection, how long the data is retained, how the data is secured, how the data is shared if it is shared, how users may access, update and delete the data, and any other choices that users have with respect to the data. Browsers typically offer to remember the user's decision for future sessions, along with a means to revoke permissions.

Other examples include media capture via HTML forms, media streams and image capture via a camera. An extension to HTML forms allows the browser to prompt the user to select a media file from the local file system and upload it as part of the form submission process. The action taken by the user to select the media file is taken as implicit consent for uploading the file. W3C is also working on APIs for taking a photo, or streaming audio or video from the device's microphone and camera. These are handled in a similar way to the geolocation API in that the request by a script to use these features results in the browser asking the user for permission.

The Full Screen API allows apps to present in fullscreen mode. After entering fullscreen mode, the user is made aware that the presentation is full screen, and given a chance to revoke the permission. The specification states:

> *User agents should ensure, e.g. by means of an overlay, that the end user is aware something is displayed fullscreen. User agents should provide a means of exiting fullscreen that always works and advertise this to the user. This is to prevent a site from spoofing the end user by recreating the user agent or even operating system environment when fullscreen.*

See explanation by Chris Pearce.

The W3C Device APIs Working Group previously worked on a proposed means for standardizing names for permissions for given APIs (Permissions for Device API Access), however, this hasn't been updated since October 2010.

A.5.2. What has been done right or wrong?

Marcos Caceres has asked for a meaningful discussion of what has been done right or wrong on the Web. He cites the way the Fullscreen api's permission works, and how geolocation API works the same across the Web and iOS. The same with WebRTC and other permissions dialogs we encounter in browsers and how users manage those (e.g., pointer lock).

A.5.3. Boris Smus on installable hosted web apps

Boris Smus' blog on Installable web apps: extend the sandbox argues for installable server hosted apps with a clear "installation" step where apps can ask for additional permissions. The step also associates the app with an icon, e.g. on the user's home screen, that can be used to launch the app, review and revoke its permissions. He further suggests an API for installing a web page as an app. This would only work if the current execution thread is the result of a user action, e.g. clicking/tapping on a button.

```
var button = document.querySelector('button#install');
button.addEventListener('click', window.app.requestInstall);
```

He also proposes an API for apps to request permissions at install time:

```
window.app.requestInstall({permissions: ['audioCapture']});
```

A.5.4. Robert O'Callahan on Permissions For Web Applications

Robert O'Callahan's blog on Permissions For Web Applications argues against introducing Android-like bundling of permissions with "up front" permission grants. Instead, he encourages:

- **Implicit Permission Grants** -- where the user action can be taken as granting permission
- **Ask Forgiveness, Not Permission** -- where the actions of a malicious application can be easily detected and safely undone
- **Remember This Decision** -- remember if the user previously refused permission
- **Permissions in Context** -- it is preferable to ask the user for permission in the context of use

A.5.5. Discussions in the System Applications Working Group

A pertinent thread of discussion on the SysApps WG archives: **Permissions UI & Necessary API**

- April 2014, starting with Doug Reeder (Wednesday, 23 April)
- May 2014, starting with Marcos Caceres (Friday, 2 May)

Doug initiated the discussion by referencing Brenden Mulligan's article on The Right Way To Ask Users For iOS Permissions. Anssi Kostiainen responds with:

> *I extracted the following recommendations that might work for the Web too. I probably missed some, so feel free to expand:*
>
> 1. *Allow the developer to associate a custom text string with the permission request.*
>
> *I observe some web-based platforms (see Firefox OS App manifest) already provide a similar mechanism, however, I'm not sure if e.g. Firefox OS uses the information in the context of use as recommended (or just for upfront grants)?*
>
> *Marcos Caceres comments: AFAIK, in FxOS they are only used by marketplace reviewers to understand why a feature is being requested by the developer - and then so the store reviewer can make an assessment about the truthiness of the claim during code review. In other words, I think the descriptions are just things that mostly serve store owners. These things are not displayed to users - the APIs access is then granted by Mozilla based on a successful review.*
>
> *Doug Reeder comments: Firefox OS requires an explanation string in the app manifest, for example:*
>
> ```
> "permissions": {
>   "geolocation": {
>     "description": "Needed for geotagging (where you wrote a memo)"
>   }
> }
> ```
>
> *IIRC, this is supposed to be displayed to the user during the install process. It is never displayed while an app is running. I'm proposing an explanation string per request, something like*
>
> ```
> navigator.geolocation.getCurrentPosition(
>   successFunc,
>   errorFunc,
> ```

```
  {timeout: 300000, description: "geotag memos"}
);
```

We (Mozilla) never prompt users to grant permissions at install time on Firefox OS. We only prompt at runtime for privacy related ones that users can understand: geolocation, sdcard access, contacts for instance.

We currently don't use the explanation string anywhere - one place we could is in the settings panel that let the user revoke or grant permissions for an app after installation.

Implications of this feature on the Web are a bit different than on native ecosystems in which the content is usually curated. For example, an evil application could lie to get you grant access to some capabilities it wants to use for other — potentially malicious or otherwise harmful — purposes than it told you to.

Doug Reeder: Colin Walters, in a comment on Robert O'Callahan's blog post (Permissions For Web Applications) points out "you have to know applications can pass messages to one another, so the permission set is in reality the union of all of the ones from any apps installed from a developer (or cooperating developers)"

Once info is passed to an app, there's no technical control over what it does with that info, only social control (reviews saying "this app lies about what it does!" ... or a consumer protection agency investigation). In the current model, the app makes no promises (other than app store boilerplate). An explanation per request allows an app to be be clear. If many apps are clear, the hope is that users will pay attention, be wary of apps that are vague, and avoid those that lie.

An explanation per request does not imply a security dialog per request. I envision the system showing one security dialog per description string. So, the user would grant permission to 'allow searchablenotes.hominidsoftware.com to use your computer's location to "geotag memos"'. Most apps & websites would use only one description, but some would use two or more different ones, allowing a separate permission for 'allow example.com to use your computer's location to "connect you to an appropriate call center"'.

Some further notes: infobars that have proliferated recently are part of the chrome, and the convention has been the web content is unable to modify this part of the UI. That said, there's precedence in legacy alert(), prompt(), and confirm() which allow the developer to customise the message shown in these system UI widgets. These *dialogs are modal, and are shown overlaid on top of the web content so they're different to infobars in that regard.*

Furthermore, in some browsers, e.g. on iOS Safari, modal dialogs are used similarly to infobars in other browsers to ask for permissions. Actually with confirm() you could pretty closely emulate the iOS Safari's "http://example.org Would Like to Use Your Foobar [Don't Allow] / [OK]" dialog only if the button labels were developer-configurable (or if there would be a confirmPermission() with labels that match the platform conventions).

2. *Prefer user-triggered dialogs.*

This reminds me of the good old <input type=file>. We've extended the model with some extra capabilities such as HTML Media Capture in the past.

Doug Reeder: This is great where you can do it, such as a standard map app, which can have a button "Show my location". I'm running into a situation where the user gets a system permission dialog, and it's not clear why to him or her. This is where it would be helpful for developers to pass a context string as part of an API request.

3. *Show an educational pre-permission overlay.*

This does not require any changes to the platform APIs. The developer can build such a dialog with HTML and friends. I'd guess there must be some examples of this type of a pre-permission overlay being used on the Web too, anyone?

Coming back to the key finding of the article. It appears the approaches outlined make sense for iOS given the significantly increased acceptance rates, so I think it is a worthwhile exercise to see whether some of this could be used for the Web too.

I asked if it makes sense to allow the developer to associate a custom text string with the permission request. However, for uncurated applications there is a risk of apps misleading users into granting permission by providing incorrect descriptions of the purposes involved.

Dave Raggett notes:

Moving the explanation to the app's content (as suggested by Brenden Mulligan for iOS apps) won't stop apps from lying, so I don't find that to be a compelling solution. It is up to reviewers and *curators of app stores to ascertain when an app is misleading users or otherwise falls foul of the app store's requirements.*

A more compelling argument is that app developers will want control of how the justification for using a given capability is presented to the user. This further suggests the requirement for apps to determine which of the following apply:

a. *user has yet to be asked for a decision*
b. *user has previously granted permission*
c. *user has explicitly denied permission*

Without this information, it would be hard for developers to provide the appropriate user experience. Does FirefoxOS offer developers this info?

p.s. if the user previously granted the permission just this time, the situation is essentially (a) in that attempting to access the capability now will result in the browser asking the user for permission.

Anssi responds:

An API that fulfils the requirements a-c above was experimented with in the Device APIs WG couple of years ago, known as Feature Permissions [1]. The spec was put on hold as "the only immediately obvious relevant use case [was] for Web Notifications". Eventually, the Web Notifications API settled on a slightly different model [2] in which the UA's default permission setting (allow or deny) is not exposed.

Some known issues with the model in [1] include privacy concerns over exposing user's preference to the web content (from the privacy perspective, the web content should not need to know whether you have explicitly declined or just ignored the permission prompt) and other potential for misuse (e.g. block the user's flow until she grants permission). That said, this thread suggests there may be also legitimate use cases for such a feature.

Doug - the API shape aside, do you think an API such as [1] would be part of the solution to your problem?

Doug responds: Yes. If my app knew the user had not granted permission (despite setting the app preference), it could open a dialog setting out choices to the user.

> *I'm wondering what are the lessons learned from the Web Notifications API that ships in Firefox, Chrome, Safari, and some others browsers. I recall reading web developers' feedback a while ago but I'm unable to find a good pointer now.*

Marcos responds to Dave:

> *In respect to a-c, it might be interesting to map these out for various APIs. For example, Geolocation:*
>
> *a. user has yet to be asked for a decision*
>
> *The developer can record this in localStorage.*
>
> ```
> localStorage.geoEnabled = "haven't asked yet".
> ```
>
> *b. user has previously granted permission*
>
> ```
> navigator.geolocation.getCurrentPosition(function(){
> if(!localStorage.geoEnabled !== "yep"){
>   localStorage.geoEnabled = "yep"
> });
> ```
>
> *c. user has explicitly denied permission*
>
> ```
> navigator.geolocation.watchPosition(function(e){},
> function(e){
>   \\1 === PERMISSION_DENIED
>   if(e.code === "1") {localStorage.geoEnabled = "denied"};
> });
> ```
>
> *So, with Geolocation you might have enough information.*
>
> *Doug: Unfortunately for my situation, if the user allows geolocation once, it says nothing about whether it will be allowed the next time. In Chrome, allowing geolocation is persistent, but in Firefox and Firefox OS it's not persistent unless the user clicks a second control.*
>
> *Marcos responds: Ok, good to know. Do you think just adding*
>
> ```
> `geolocation.permission === 'enabled'`
> ```
>
> *or similar would address the use case? Also, maybe good if we could move this back to the list?*
>
> *Doug responds: Yes, that would solve my problem.*

> *Theoretically, permission could change between checking and calling getCurrentLocation, but if I'm calling them in the same tick, ocurrence should be vastly rarer than geolocation failures.*
>
> *Anssi responds: I think that is not really a concern. I don’t see anything getting in between the following considering the single-threaded nature of JavaScript:*
>
> ```
> if (geo.permission === 'enabled') {
>   geo.getCurrentPosition();
> }
> ```
>
> *Or perhaps you have a more complete real life example in mind?*
>
> *Anssi comments: Whether implementations persist, or allow the user to persist, permission settings vary by browser and by feature. And this will likely remain so.*
>
> *To further complicate things, sometimes subtle hints of trustworthiness of the site are used to decide whether to allow persisting a permission or not. For example, Chrome allows persisting gUM permission only if the content is served over HTTPS, while for Geolocation the permission is persisted regardless of the protocol.*
>
> *iOS is probably the strictest in this regard, and never persists permissions for regular web content (only for things bookmarked to homescreen, which is another hint of trustworthiness).*
>
> *An opportunity to dig into this a bit more.*

On May 6th, Anssi wrote:

> *A proposal by Nikhil elsewhere on how Promise.all() might be used:*
>
> ```
> Promise.all([
>   Notification.requestPermission(),
>        // Some shimmed form that returns a Promise
>   navigator.push.hasPermission()
> ]).then(function(perms) {
>   if (perms[0] == 'granted') { // notifications ok; }
>   if (perms[1] == 'granted') { navigator.push.register(); }
> });
> ```
>
> *Promise.all() returns a promise that resolves when all of the promises passed to it have resolved. I’m wondering if something like the following has obvious issues:*

```
[NoInterfaceObject]
interface Permissions {
  Promise requestPermission ();
  Promise hasPermission ();
  /* ... */
};

Notification implements Permissions;
PushManager implements Permissions;
/* ... */
Geolocation implements Permissions;
```

I think not all APIs can be retrofitted with this, but many could.

Marcos responds: Seems unnecessary to have this return a promise, IMO. Just make it an attribute.

Anssi then says: Yeah, the reason for that was to make it work with Promise.all as suggested by Nikhil. However, it seems whatever passed to Promise.all is converted to a promise by means of Promise.cast, so we could make it an attribute as you suggest.

Details aside, I think the main question is does such an interface make sense in the first place?

Marcos then says: Like I said previously, I think the only way to know is to work through some example cases with real code. Doing thought experiments can only get us so far. We would also need to find a few more example cases in the wild and then we can take those to the appropriate WGs.

Anssi: I actually already asked Nikhil in the GH issue from where this idea originated from whether he has been doing further exploration in code. If someone comes up with other experiments, please let us know.

Marcos: I'd be inclined to prototype having an attribute in Gecko. The `requestPermission()` method seems redundant to me because interfaces already implicitly or explicitly have these methods (yes, they are inconsistent across the various APIs - but in the case of Geolocation the permission request is explicitly bound to an action - watchPosition(), getCurrentPosition(), and I think that is a "good thing"[tm]).

Some other APIs - specially new ones - would likely benefit from `requestPermission()` tho, but I'm still not sure if existing APIs would.

Anssi: It appears some implementers would prefer to have hasPermission() return a Promise, at least for some APIs, see Peter Beverloo's email of 29 Apr 2014:

> *The Notification specification defines a static Notification.permission accessor, which returns one of {granted, denied, default}. This requires the browser to synchronously determine whether the page has permission to show notifications, whereas checking this may be an asynchronous operation. This is the case in Chrome.*
>
> *Before this becomes a paradigm, could we consider having a static hasPermission() instead, returning a Promise?*
>
> *I'll add a UseCounter to Blink for tracking Notification.permission usage, but it will take some time before conclusive usage data comes in.*

Chrome is collecting stats on the usage of the sync .permission, so we should get some data on how widespread the usage is.

A.6. SUMMARY AND FUTURE WORK

As the Open Web Platform expands, new capabilities are likely to require new ways of managing permissions. Some platforms, e.g. Android, ask users upfront for permission when an app is installed or first run, whereas others like iOS ask users for permission when the application is attempting to use a given capability. Rather than asking the user for permission in advance, another approach is to invite the user to continue or to cancel an action after it has occurred, i.e. asking for forgiveness rather than permission -- this is the approach taken in the Fullscreen API, see the explanation by Chris Pearce. In some cases, the user's actions can be taken as implicitly granting permission, for a detailed analysis of this approach, see Roesner et al. A further approach is for users to delegate decisions on permissions to a trusted 3rd party (if it's okay by them, it's okay by me). What is needed to arrive at a consensus for a cross vendor solution?

A.6.1. Some questions for further study

Group 1: User Consent

- What criteria are there for choosing between asking users upfront for permissions, asking at the time of use, asking for forgiveness rather than permission, implicit approval via user actions, or mediated by the platform or trusted 3rd party?

- Is it practical to offer users clear explanations about how capabilities will be used prior to being asked to make decisions on whether or not to grant applications permissions for these capabilities? What is the basis for users to trust these explanations?
- How can developers tailor the user experience, e.g. the presentation of justifications of the need for a given capability prior to asking the user for permission? This necessitates a means for apps to discover whether the user has previously granted permission, has previously declined to grant permission or has yet to be asked.
    - Standards are needed for common capabilities and how these are named in order to provide developers with good expectations of interoperability, and to give users a consistent and understandable experience across different applications and devices.
- What are the implications that different approaches have for API design? For instance, the implicit approach could enable APIs only in execution contexts triggered by user interaction, e.g. a click, tap or keyboard short cut. This, however, is open to abuse by mislabeling UI controls to fool users to into initiating actions without their knowledge.
    - What are the implications for app developers when the user has granted some but not all permissions requested by an app? This relates to proposals for returning a promise when testing if permissions have been granted.
- Can we reach a consensus on a consistent approach to whether users are offered the chance to apply their decision for the rest of this session and subsequent sessions? This should depend on the level of trust in the application, e.g. whether it was accessed over an encrypted connection, whether it is from a trusted website, whether it has a high reputation, and so forth.

Group 2: Delegated Trust

- What roles are there for trust delegation? This could, for example, be exploited to avoid asking users for permissions for capabilities that are hard to explain. Apps stores may be trusted on the basis of their vetting procedures. Well known websites, that host their own apps, may be trusted on the basis of their brand. Other sites may find it advantageous to have their apps endorsed by a trusted third party. What kinds of limits can be imposed for trust delegation? For instance, limiting delegation to given categories of apps, and excluding apps featuring in-app payments.
- Web apps have the unique ability to embed one another (through e.g. iframe); how does this embedding affect the ability of a Web app to request permission? how does an app with privileges indicates its willingness to be used with privileges once embedded? how does an app prevent an embedded app to request privileges (e.g. with the sandbox attribute of an iframe)?

Group 3: Permission Management

- When and how does the user know that a Web site has access to a given permission? how to deal with indicators in the browser chrome on mobile screens (with limited screen real estate), or when they app is running full screen?
- Is there a need to inform apps when the user revokes a permission, e.g. to enable the app to dynamically adapt the user experience to match?
- How should browsers adapt their permission models to the security environment of the app? How does the use of HTTPS, Content-Security-Policy, script integrity, cross-origin requests affect the permissions an app may be granted?
    - with the emergence of ServiceWorker, a number of Web app will be able to run in the background; how does the operation in background/foreground of an app affects its privileges?
    - some mobile platforms restrict some permissions to certified apps, to which other apps need to delegate the privileged operation; how does that model apply to the Web? How does it relate with proposals around integration of third-party apps such as the late Web intents, or registerProtocollHandler()
- What level of granularity is appropriate for permissions? Too low a level will make it hard to explain to users, whilst too high a level will limit the end user's freedom of choice. Furthermore, the granularity of permissions will have consequences for developers in how they deal with the cases where users decline to grant particular permissions.
- Some capabilities are very specific to a given platform and as such are inappropriate for standardization. Conversely, there is pressure to agree on a standard where many developers are seeking a cross-vendor solution.
- Access to some capabilities may be restricted, e.g. consider the case where an application can only access the engine related API in a car if the application is signed by the car manufacturer.

Group 4: Miscellaneous Topics

- How much leeway should be left to individual implementors of the Open Web Platform? For example, does it make sense for the application manifest to list the permissions needed by the application, but leave it up to the platform implementation to determine whether to ask upfront or at the time of use?
- It may take some time to arrive at a consensus for a detailed solution, so can we reach some initial agreements that enable work to proceed in parallel on standards for APIs for specific capabilities?

An open face to face meeting is now planned for early September 2014 to bring a variety of stakeholders together to discuss trust and permissions, and to try to determine a roadmap towards a broad consensus. One possibility would be to set up a W3C Community Group to continue the discussion with a view to feeding into the standards track with a chartered work item in a W3C Working

Group. If there is a strong consensus on the approach to be taken, then we could skip the Community Group step and go straight to the Working Group, or perhaps to set up a Cross Working Group Task Force. Note that we need the agreement of sufficient browser vendors to ensure that the work is widely deployed.

At the end of the meeting we would like to have a clear idea for next steps:

- Areas where we have a rough consensus and need to work on the details?
- Areas where we are still some way apart and what steps can be taken to close the gap?
- What can be ruled as out of scope for W3C work on trust and permissions in the open web platform?
- Whether we want to set up a Community Group, a Cross Working Group Task Force or some other approach?
- What design rules can we give to allow work to proceed in parallel on APIs and trust & permissions?

Some further reading:

Note: this is just a sample, and not intended to be authoratative.

- Information on API Permissions collected by the Web and Mobile Interest Group
- How to Ask For Permission, HotSec'12, Porter Felt, et al.
- Towards Comprehensible and Effective Permission Systems, Adrienne Porter Felt, Ph.D dissertation
- User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems, Roesner et al
- Exploring Notice and Choice: Design Guidelines for a User-Centered Permission Model for Personalized Services, Johnson et al
- Installable Webapps: Extend the Sandbox by Boris Smus
- Permissions For Web Applications by Robert O'Callahan
- Draft white paper on trust and permissions by Dave Raggett

A.7. ACKNOWLEDGEMENTS

This work was done with support from the European Commission under grant agreement no: 611327 (HTML5 for Apps: Closing the Gaps). Thanks also to Anssi Kostiainen, Marcos Caceres and Dominique Hazael-Massieux for their comments on an earlier draft of this document.

A.8. APPENDICES

This section contains material that has been moved from earlier sections. This includes cross platform frameworks where the permissions model is inherited from each of the target platforms. It also includes libraries for web applications where the permissions model is inherited from the web platform.

Cross platform frameworks allow developers to create app using cross platform technologies, and either compile them into native apps that can run on a variety of operating systems, or provide a native run-time that can execute cross platform code. This can cut the time to deliver to multiple target platforms. Cross platform frameworks inherit the trust/permissions model from the native platform.

According to Research Guidance (see below), the most popular tools are PhoneGap and jQuery Mobile, followed by Adobe Air, Qt Creator, Unity 3D, Titanium, Marmalade, Sencha Touch, Xamarin, Unity Mobile, and Corona SDK. Cross platforms tools are mainly used to develop apps for games, followed by utilities, business, education and entertainment in decreasing order of popularity.

For more in-depth reviews of cross platform frameworks, see:

- Cross Platform App Development Tool Benchmarking 2013 , October 2013, Research Guidance
- In-depth profiles for major cross-platform developer tool vendors, February 2012, VisionMobile.
- Pros and Cons of the Top 5 Cross-Platform Tools, November 2013, DeveloperEconomics

A.8.1. Telerik

Telerik provides UI controls for HTML5 and .NET, along with the Telerik platform for mobile app development.

A.8.2. Appcelerator Titanium

Appcelerator is a Californian technology company that provides the Appcelerator platform for cross platform mobile development with an open source Titanium SDK and an enterprise software suite covering development, testing, deployment and analytics. Titanium supports iOS, Android, Windows Phone, Blackberry OS and Tizen, enabling developers to create rich native mobile apps from a single JavaScript-based SDK. Titanum includes Alloy which features a model-view-controller architecture for rapid development of UI components based upon XML markup and style sheets, as a declarative alternative to creating UI components directly from JavaScript.

A.8.3. Xamarin/Mono

Xamarin enables developers to create apps for iOS, Android, and Windows Phone using the C# programming language, and derives from earlier work on the Mono open source project to support Microsoft's .NET framework on the Linux operating system. Xamarin includes binding for the indigenous SDKs for each of the platforms it supports. Where needed, you can directly invoke Objective-C, Java C and C++ libraries. Xamarin claim that applications can share up to 90% of their code across platforms when using the Xamarin Mobile library. Further details are given in the Xamarin developer guides.

A.8.4. Qt

Qt features a cross platform integrated development environment (Qt Creator) and run-time with support for C++, and the JavaScript like QML user interface modelling language. Qt targets desktop, mobile and embedded devices, e.g. Android, BlackBerry, iOS, Linux/X11, Mac OS X, Windows and Windows CE. Qt Cloud Services provides support for application backends. More details are available on the Qt Project website.

A.8.5. Unity 3D/Mobile

Unity is a cross platform game engine for web plugins, desktop, consoles and mobile devices. For scripting you can use UnityScript (similar to JavaScript) or Boo (similar to Python). Supported platforms include BlackBerry, Windows, Windows Phone, Mac OS X, iOS, Android, Adobe Flash, PlayStation, Xbox, and Wii.

A.8.6. Corona SDK

Corona SDK enables developers to create 2D games, business, eBooks and educational apps for mobile devices including iPhone, iPad, and Android. The SDK includes a wide range of third party tools and services. Scripting is based upon the Lua programing language.

A.8.7. Marmalade

The Marmalade SDK targets desktop, tablets, smart phones and TVs. Developers can work with C++, Lua, HTML or Objective-C. Apps are compiled to native binaries for ARM or x86 and combined with a platform specific loader. Target platforms include Windows, Mac OSX, LG Smart TV, ROKU, iOS, Android, BlackBerry, Windows Phone and Tizen.

A.8.8. GINGEE

GINGEE provides for cross platform development on mobile devices with a focus on games. Its Liquid UI adapts the UI to the device allowing apps to have the same look across all devices, based upon a library of UI components. The SDK claimes to offer near-native run-time speeds, GINGEE offers easy integration with social media, analytics and monetization. Targetted platforms include iOS, Android, Amazon Kindle Fire, Barnes & Noble NOOK, BlackBerry, Facebook, Smart TV and Windows PC.

A.8.9. Codename One

Codename One is an open source project that enables developers to use Java to create apps that have the native look and feel for iOS, Android, Windows Phone and Blackberry.

A.8.10. DragonRad

The DragonRAD Designer is developer tool for creating enterprise apps for smart phones and tablets BlackBerry, Android, iOS and Windows Mobile. It supports the Lua scripting language. The DragonRAD Host provides access to enterprise databases via a Windows or Linux server. The DragonRAD Client provides a native run-time that interprets and runs your application.

A.8.11. RunRev LiveCode

LiveCode is inspired by Apple Hypercard with a drag and drop developer tool and a scripting language resembling Hypercard's HyperTalk, see the beginner's guide. The component model includes stacks, cards and objects, and is event driven. LiveCode stacks can be build for Windows, Mac OS, Linux, iOS and Android and inherit the native platform's look and feel.

A.8.12. IBM Worklight

IBM Worklight provides a framework for developing, running and managing HTML5, hybrid and native apps on smart phones and tablets. You develop with HTML5 and JavaScript and then customize the resources for each target platform. The framework can generate web and native code specific to each target environment, For hybrid apps, Worklight relies on PhoneGap for access to device APIs. You can also take advantge of third party tools, libraries and frameworks including jQuery Mobile, Sencha Touch and DojoMobile.

A.8.13. MoSynch

MoSynch is an open source SDK that allows you to develop mobile apps using C/C++ and HTML5/CSS/JavaScript with a native look and feel for each target platform, including Android, iOS, Windows Mobile, Windows Phone, Symbian, Java Mobile and the Moblin platform.

A.8.14. RhoMobile

This is an integrated framework based upon the Ruby programming language.

Motorola Solutions' Rhomobile is an open source framework for developing native apps for smartphones including iOS, Android, BlackBerry, Windows Mobile and Windows Phone. Rhohub is a service for developing apps online. Developers can use JavaScript or Ruby scripting languages with the Rhodes API for access to device level capabilities including the camera and device location. Rhomobile is similar to PHP in allowing you to create user interfaces using HTML5 markup with embeded code that can be used to tailor the user experience to the target platform.

A.8.15. Whoop

The Whoop Creative Studio is a drag and drop design tool for easy development of mobile apps for iOS, Android, BlackBerry and Windows Phone. It seems to have been discontinued.

A.8.16. WAC 2.0

The former Wholesale Applications Community (WAC), together with mobile network operators (Carriers), developed a suite of specifications for mobile web applications. These were subsequently transferred to the GSMA in July 2012 when WAC was dissolved. WAC built upon the W3C specifications for HTML5 and the BONDI project of the preceding Open Mobile Terminal Platform Ltd.

WAC had the aim of enabling developers to create packaged web apps for use on home screens and app stores. The packaging format uses W3C Widgets, Access control policies are based upon rules expressed in the XML-based XACML rule language. According to wikipedia, Policies can be set on a widget provider level (for signed widgets) on a widget level or on an API call-by-call level for web pages.

I've asked Nick Allott for further information on the approach taken for user consent, and any feedback gathered from developers and end-users on its effectiveness.

A.8.17. jQuery

jQuery is a very popular cross brower JavaScript library that simplifies application development, for instance, network access, manipulating the DOM, and applying animations and effects. There is a growing set of plugins for extending jQuery. jQuery Mobile is a related cross browser JavaScript library built on top of jQuery Core, and aimed at making it easier to create responsive websites for smart phones, tablets and desktop. It supports a range of UI controls and themes. Both libraries stick within the existing browser security model.

A.8.18. Sencha Touch

This is a UI framework for mobile devices using a JavaScript library, and enables developers to develop mobile web applications that look and feel like native applications. Sencha Touch targets, iOS, Android, Windows, Tizen and Blackberry based devices.

A.8.19. Dojo Mobile

A.8.20. Netbiscuits Tactile

Netbiscuits Tactile is a cloud software service for cross-platform development, publication and monetization of mobile sites and apps based upon HTML5. It looks like Netbiscuits has now discontinued this service.

A.8.21. WidgetPad

This is an open source platform for developing HTML5 apps tailored for devices using Android, WebOS and iOS. It seems to have been discontinued.

A.8.22. webinos

An EU research project that focused on extending the Open Web Platform with a rich suite of APIs for accessing resources within devices that form a user's Personal Zone, e.g. a desktop PC, a tablet, a smart phone, a smart TV or a web-enabled car. The devices enrolled in a Personal Zone are subject to strong security with mutual authentication, and access control policies based upon XACML.

B. MINUTES FROM MEETING ON TRUST AND PERMISSIONS FOR WEB APPLICATIONS

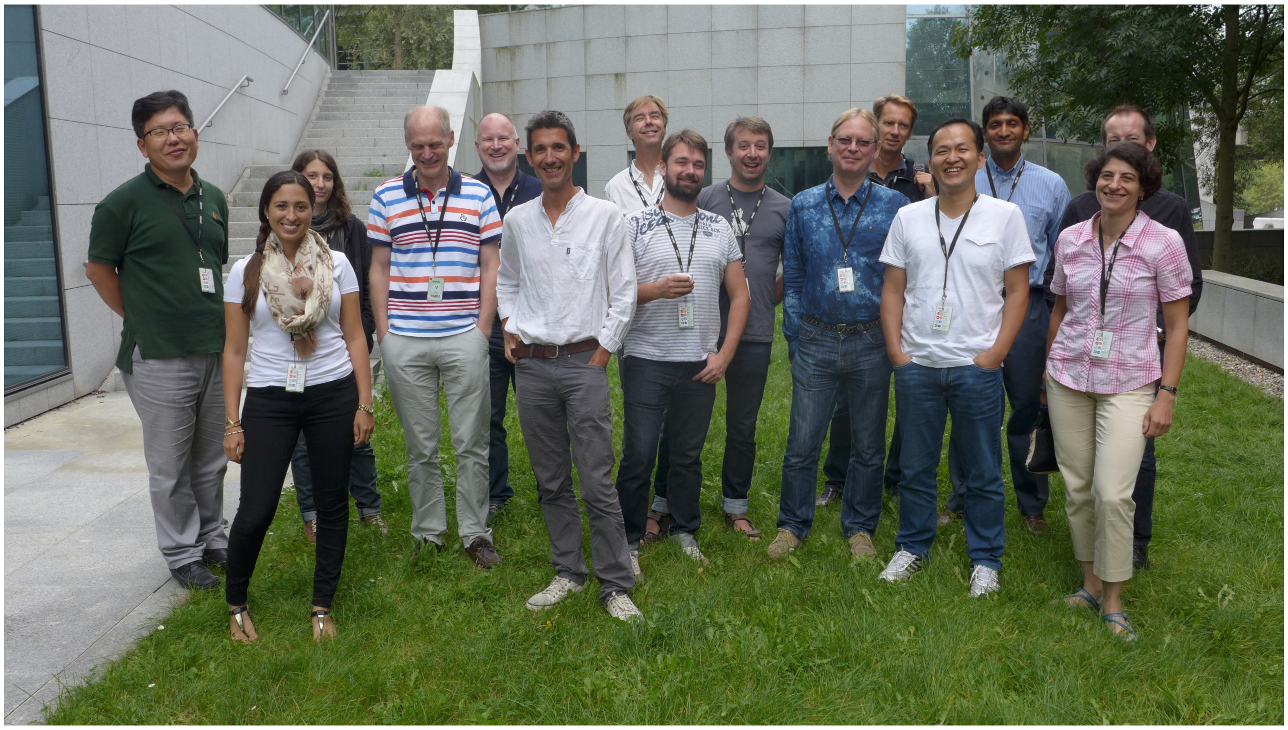
3–4 September 2014, Paris, France

Meeting page

Present:

- Dave Raggett, W3C
- Dominique Hazaël-Massieux, W3C (remote)
- Robin Berjon, W3C (webapps etc.)
- Wendy Seltzer, W3C (security, privacy) 2nd day only
- Stefan Håkansson, Ericsson (Co-chair Media Capture TF, WebRTC)
- Philipp Hoschka, W3C
- Giridhar Mandyam, Qualcomm
- Claes Nilsson, Sony Mobile
- Wonsuk Lee, Samsung
- Vadim Draluk, GM (automotive)
- Adrienne Porter Felt, Google
- Jonghong (Jonathan) Jeon, ETRI
- Steven Woolcock, Apple
- John Hazen, Microsoft
- Stephanie Ouillon, Mozilla
- Kenneth Rohde Christiansen, Intel
- Olivier Potonniee, Gemalto

- Anssi Kostiainen (remote)
- Virginie Galindo (remote)


Credits to Johnathon Jeon

B.1. WEDNESDAY, 3RD SEPTEMBER

B.1.1. Session 1: Introductions by participants

We went around the room (and the phone) introducing ourselves.

Dave set out the meeting objectives:

- Areas where we have a rough consensus and need to work on the details?
- Areas where we are still some way apart and what steps can be taken to close the gap?
- What can be ruled as out of scope for W3C work on trust and permissions in the open web platform?
- Whether we want to set up a Community Group, a Cross Working Group Task Force or some other approach?
- What design rules can we give to allow work to proceed in parallel on APIs and trust & permissions?

At the end of the meeting we would like to have a clear idea for next steps

B.1.2. Session 2: Logistics and agenda tweaking

We decided to stick with the agenda on the meeting page

Session 3: Review of existing practices for the Open Web Platform (OWP)

Wonsuk goes through history from SysApps till today:

SysApps attempted to handle many APIs (around 15) with focus on packaged apps with input from Tizen, Firefox OS and Chrome OS.

Issues: Packaged apps were very tied to the specific stores which broke the basic idea of the web: the hyperlink. Second issue: Versioned apps, web apps are usually updated on the go without the users having to install an update. Less than 5% of apps on the FirefoxOS market place use APIs requiring the app to be a packaged app instead of a hosted one.

Note: It was decided that content protection (DRM) would be considered out-of-scope for this meeting.

From GMandyam[a]: I believe the reason the SysApps WG failed in developing a runtime and security specification for installable web apps was that many platforms had already solved this problem[b] independent of the W3C. Any standard developed by the W3C in this area may be ignored by the industry. The focus for this work should be on hosted web apps.

Geolocation-specific issues (GMandyam):

The PING group conveyed the concern (during the May meeting focused on the re-chartered Geolocation Working Group) that service providers do not provide information to the end user as to how geolocation information is used by the website. The current browser permissions model does not cover this. However, mechanisms that rely on the browser chrome for expressing intended use of geolocation information are subject to abuse.

Geolocation WG looks at adding geofencing which is security sensitive (indoor location, SSIA sharing, etc). Ad services: Sites adds the location as part of the request. There were privacy concerns about setting up geofences with respect to indoor centroids, as many times the most effective way to define the centroid is with respect to an indoor WiFi AP. Does the end user want a website to know the SSID of an AP in his/her home? Does the browser need to obscure such information from the website?

Native platforms have solved geolocation permissions mainly through declared permissions[c] (e.g. in a manifest) and/or application signing. The signing is supposed to be an indication of trust - the app has been vetted by an app store. However, app stores still have to rely on end user feedback in order to revoke permissions for rogue apps.

Media Capture/WebRTC (StefanH):

Pioneered by Ian Hickson, first proposal was device element, based on feedback the navigator.getUserMedia was introduced. The original idea was that the app should not know anything before granted access (the user should be presented with the possibility to opt in to any devices (=microphones, cameras) available, but the app would not know about available devices before access granted), but has developed into that the app _is_ able to find out what devices are available before asking access, and can ask for what devices it wants access to. In short, the current model allows for more fingerprinting than the original model.

Persistent access has also been introduced, meaning that an app could get access (if the user agrees) to devices without being presented with a prompt. This requires that the app is served over https.

Once the app has access to devices it can record to file (and then send the recording anywhere), capture images, send to arbitrary peer. But isolated media streams have been introduced, they prohibit recording/image capture and can only be sent to a peer with a defined identity.

Screen sharing is currently being discussed, proposal at http://rawgit.com/fluffy/w3c-screen-share/master/screenshare.html

Dom about the DAP models:

4 of DAP APIs are permission-less: Battery Status API, Ambient Light Event, Proximity Events, Vibration API. HTML Media Capture uses an implicit approval model: by taking the picture and selecting it, the user implicitly approves sharing it with the browser app. Network Service Discovery has a prompt then implicit-selector to determine from which devices to e.g. select media. The upcoming Wake Lock API expects to use the “ask for forgiveness rather than permission” approach, whereby the ability to keep the screen up is granted by default, but the user gets informed and can revoke that permission at the same time. Web Intents waking up again

B.1.3. Session 4: Review of approaches used by other platforms (web and native)

We had summaries for iOS, Android, Windows Phone, Chrome Apps, Firefox OS, and GM

Steven Woolcock gave us a quick overview of Apple's approach for iOS. There are a small number of permissions and when apps invoke the APIs for the corresponding capabilities, the user is prompted to obtain the permission. The context in which the prompts occur make it easier for users to understand what the prompts are for as compared to asking for permissions at install time. Apple has recently provided a means for developers to pass an explanatory string for use in the operating system generated permission dialogues.

Steven noted that in certain cases it would be desirable to ask for permissions prior to use. One example is where parents want to control what permissions are appropriate for apps used by their children.

Adrienne Felt summarised how permissions are used in Android. There are something like 150 permissions available. The ones that developers think they will need have to be declared in the application manifest and users have to give their consent as a precondition for installing an app. A few permissions are only available to applications certified by Google. Dom cited the API for bricking a stolen phone as an example.

She also described a study she had been involved in which exampled app program code to see which APIs the app used and compared this with the manifest to see whether developers are requesting privileges they aren't using. For the most part developers just request the permissions they need, and most of the exceptions seem to be explained by misunderstandings of what a given permission is required for.

Adrienne commented that when asked users thought about pernissions in terms of what they thought developers would do, and by and large, didn't appreciate the space of possibilities that permissions enable.

Dave wondered if permissions could be tied to use policies that are contractually binding on developers. If apps are found to breach these policies, then the apps and developers could be black listed.

Others noted that the Android permission model trained users to click through the consent dialogue in order to try out the newly installed app.

In many ways users would prefer to know whether the app is trustworthy or not and not have to see the legalese of the permissions dialogue.

John Hazen: Notes on Microsoft Permissions Model

The Microsoft model for permissions in Windows 8 was driven by earlier experiences with User Account Control (UAC) prompts, and by a desire to minimize the total number of permissions available to developers and end users. In both of these cases our prior experience indicated to us that users and developers will do the wrong thing, either by simply clicking through dialogs or by overdeclaring permissions just to make the app work. Our desire to keep the list short was driven by a desire to ensure that the developers and users could reason clearly about the capability. We ended up with about a dozen capabilities, but they are not very nuanced. For example, the picture library capability does not distinguish between read-only access or read-write access. The desire to minimize prompts led us to favor implicit consent through user actions, for example using the file picker to access particular files, or even choosing to install a particular app from the Windows Store. The only place where we decided to keep explicit prompts was for capabilities which expose something about the user in real time (geoloc, camera, microphone, etc.). In most cases, the end-user is delegating trust to the Store for many of the capabilities. This of course means that the Store needs to have an adequate review process to weed out most malware, but also needs to have revocation capabilities for cases where mis-behaving apps have been released to the wild. Finally, I also noted that it is not always the end-user who really should be making the trust decision--for example, in the Windows ecosystem access to Enterprise Credentials (Domain creds) is a powerful capability, but the resources that are being protected do not belong to the user - they belong to the network administrator, so the platform allows the network admins to allow/deny such apps from running on their networks.

Stéphanie Ouillon: Firefox OS Permission Model

We worked on a paper about the app permission model in Firefox OS. The following is copied from that document except for the table on permssions vs levels.

Permission Model Overview

Firefox OS's proprietary APIs require permissions if they expose sensitive data or functionality to the web. Apps declare the permissions they want in an associated JSON manifest. Permissions are then denied/allowed or granted by the user according to the permission level of the app. Additionally permissions which have privacy implications also require user consent, by way of a runtime at first use of the API.

Permission Levels

"Web" apps permissions

"Web" apps follow the web security model and in general can not gain higher privileges than regular web pages. They may be hosted or packaged, and installed via the Firefox Marketplace, or from any website.

"Privileged" permissions

More sensitive APIs are restricted to "privileged" apps. For actually receiving these permissions, the app must be signed by the Mozilla Marketplace. "Privileged" apps are ZIP files that are digitally signed and distributed from the Mozilla Marketplace. All privileged apps undergo a security review by a (human) reviewer. The application review process is described at the following link: https://developer.mozilla.org/en-US/Marketplace/Publishing/Marketplace_review_criteria

"Certified" permissions

"Certified" permissions give access to system APIs which are needed to build a web-based mobile operating system. For security and/or privacy reasons, these APIs cannot be exposed to either privileged or hosted apps. Certified permissions can only be granted at build time (Gaia apps) and are not available to apps installed by the user (except to side-loaded apps for developers).

Challenge: exposing these APIs to the Web

Mozilla wants to evolve our proprietary APIs so they can be exposed to the web. The biggest challenge for this task is resolving intersection between the traditional use cases of native apps and the security risks of the APIs needed to support these use cases. In general there are three approaches to this problem:

- limit the functionality of the APIs so they are safe enough for the web
- adding system level mitigating controls to address security threats (e.g. notification UI to make the user aware that an app is using a certain device)
- mandating security requirements for apps to increase the level of in web application code

Firefox OS currently employs a mixture of these approaches. Making the APIs safer

- Adding more granularity to the permission model in order to expose safer API subsets

e.g the Mobile Network API exposes a read-only subset of the Mobile Connection API to allow privileged apps identify the carrier without needed to talk directly to the SIM card or mobile network. See details here:

https://developer.mozilla.org/en-US/docs/Web/API/MozMobileNetworkInfo

- Exposing an API through system-mediated UI

e.g. the Mobile ID API allows an app to verify the user’s mobile number without needing access to directly be able to send a silent SMS (which is a common use case).

See details here: https://wiki.mozilla.org/WebAPI/MobileIdentity

- Requiring user-interaction to prevent inadvertent access to APIs

e.g. Sensor access (accelerometer etc) is limited to content which is in the foreground to prevent surreptitious monitoring.

e.g. getUserMedia, as on desktop browsers, requires prompt prior to granting access to the camera/microphone streams.

System Security Controls

- Notification UI

e.g. The notification bar shows when an app is using the camera through getUserMedia, to make the user aware that the camera/microphone is enabled.

- Global configuration

e.g. Enable/disable GPS button to allow users control over when geolocation is available.

Improving trust

Enforcing an integrity mechanism to that apps can not be modified

e.g. Privileged apps require apps to be static code within a ZIP file, and a Content Security Policy is enforced to ensure that only script from within that package can be run within the privileged application origin.

Challenge: the Web is dynamic, and forcing static content limits how web apps can be developed and deployed.

- Signing application content

Privileged apps are required to be signed by the Mozilla marketplace to ensure they have gone through security review.

Challenge: current implementation requires code to be static ZIP files, similar issues as to above.

Challenge: Firefox Marketplace is the only source of trust for signing privileged applications. How do we delegate this trust to in the decentralised model of the web?

See discussion about the future of packaged apps and app developer signing: https://groups.google.com/forum/#!topic/mozilla.dev.webapi/68I80XiipBI

Reviewing application code

e.g. Privileged apps go through manual review prior to being granted privileged permissions.

Challenge: how to scale this control to the web?

API Case Studies

Advancing the permission model will require reviewing specific APIs and use cases. Below are some key APIs which represent some of the main types of APIs indicative:

API: MozMobileMessageManager
Description: API used by the system apps to send & receive SMS and MMS messages.

Notes:

- Extremely high threat API - abuse directly costs the user’s money and compromises privacy

- WebAPI Security Discussion: Web SMS API

API: DeviceStorage
Description: Add, read, or modify media files stored on the device.

Notes:

- If content is exposed to the web, it is compromised if the app developer server is hacked.
- A system-mediated UI file picker doesn’t solve the case when a gallery app wants to access *all • photos.

API: SystemXHR
Description: Allows anonymous (no cookies) cross-origin XHR without the target site having CORS enabled.

Notes:

- Risk is leading attacks on websites behind firewalls, when the phone is on an internal network.
- Whitelisting domains in the review process (too complicated for the user to approve).
- Ideally, WebSocket, WebRTC and CORS should replace TCP/UDPSocket and SystemXHR.

The following APIs are either privileged or certified but work is underway to try to expose (parts of them) to the web based on partner and app developer requests for functionality.

Work in progress

Mozilla is currently working to expose the following APIs to Privileged Apps:

- Devices
- NFC
- Bluetooth
- Wi-Fi (requested, but not priority)
- WebSerial (requested, but not priority)
- App embedding
- Lockscreen (to allow for replaceable lockscreens)
- Homescreen-webapps-manage (to allow for replaceable homescreens)
- Firefox-accounts (allow for device wide sign-in)

Other controls are currently in progress or in discussion:

- Allowing user auditing of API access:
    - One feature which is currently in progress is allowing users to monitor data usage on a per-application basis. This control could be employed for any API where resource consumption is a security concern (disk space, battery etc).

- Improving the UX by giving the user more power and information by:
    - Tying permissions to actual actions which would be meaningful for the user to better inform about the impact of granting the permission requested by an app.

    - Requiring SSL as a mandatory control.

Claes Nilsson: Trusted hosted apps for FFOS - Sony research project

The dynamic nature of hosted web apps compared to installed web apps is advantageous in many aspects. Sony is evaluating how sensitive APIs could be opened up not only for installed privileged and pre-installed certified FFOS apps but also for hosted web apps. The model is based on using the existing web security mechanisms secure transport and Content Security Policies (CSP).

The model is described below:

1. Content and manifest are downloaded to the device with ssl/tls, i.e. server authentication, encryption and integrity protection.

2. Certificate pinning is used, which means that only certain specific certificates are trusted, for example only certificates signed by Sony and Mozilla are trusted.

3. Signature of trusted app's manifest must be verified by a trusted authority

4. There is a CSP field in the manifest file. It is defined so that by default only "self" is allowed, i.e. only content from the origin domain of the app is allowed to download. It is also possible to whitelist other domains with CSP that may be accessed through ssl/tls with certificate pinning. The resources that are accessed through ssl/tls with pinning are script and style src resources. (as declared in 'style-src' and 'script-src' directives in CSP element of the manifest.)

   In addition default security policies apply:

    - Eval and related functions are disabled
    - Inline JavaScript will not be executed

5. Trusted hosted apps are allowed to access a set of more sensitive APIs than normal hosted apps.

   Manifest example:

```
{
   "name": ”Test app",
   "description": ”Test of trusted hosted apps",
   "version": "1.0",
   "type": "trusted",
   "launch_path": "/index.html",
   "icons": { "16": "/favicon.ico" },
   "developer”: { "name": ”Developer Team",
                  "url": "http://devloper.com" },
   "csp" : "script-src ’self’ https://.123.testfront.net;
            style-src 'self’ https://123.testfront.net",
   "permissions": {
          "device-storage:videos":{ "access": "readonly" },
          "device-storage:pictures":{ "access": "readwrite" }
   },
   "appcache_path": "/manifest.appcache"
}
```

Mozilla permissable table example (“trusted” is the additional application type “trusted hosted web applications)::

```
"device-storage:pictures": {
   app: DENY_ACTION,
   trusted: PROMPT_ACTION,
   privileged: PROMPT_ACTION,
   certified: ALLOW_ACTION,
   access: ["read", "write", "create"]
},
"device-storage:videos": {
   app: DENY_ACTION,
   trusted: PROMPT_ACTION,
   privileged: PROMPT_ACTION,
   certified: ALLOW_ACTION,
   access: ["read", "write", "create"]
},
```

Further information about the project

Stefan Håkansson: “Trusted Service Worker/Provider“ - Ericsson research concept

Gives regular web apps access to extended functionality. Extended functionality requires access to privileged APIs (not exposable to web application for security reasons).

A Provider runs trusted code (JavaScript) from a third party, it has access to selected privileged APIs and can formulate consent questions that make sense to users. It executes in a separate context and exposes its own API towards web applications using web messaging (postMessage).

Enables new APIs to be built on top of privileged APIs, in a way that allows them to be securely exposed to web apps.

A Provider runs in a new type of Worker - ProviderWorker and exposes its own API towards web applications via web messaging. Differs from Web Workers in that it:

- Can only execute code from trusted sources
- Gets its origin from the Provider publisher
- Access to selected privileged APIs

A manifest is used (Title, description, permissions, …)

The Provider (having access to privileged APIs) must be distributed by an app store (code reviewed etc.)

More details are in Stefan's presentation on Web Providers.

Vadim Draluk: GM focus on web and automotive:

While General Motors had announced its own HTML-based development platform a while back, strategically it is looking forward to emergence of a more standardized environment for Web applications that would facilitate build-up of an ecosystem suitable for automotive environment, among other verticals. As a result, GM has joined the W3C Automotive Business Group, and has been active in its proceedings. One of the issues that came up within that group was permissions and security model. The group decided that it would be highly beneficial to have such a model defined within W3C horizontally, such as the same one could apply to all verticals, automotive included.

There are some aspects that differentiate automotive industry requirements. First, the most typical deployment models are installed and installed-projected (that is, installed on a phone that is projecting its screen onto the IVI display), less so hosted. The reason is that, mostly due to security concerns, browsers are not installed in the IVI systems. However, attracting Web developers remains an important objective, hence the bias towards installed JavaScript/HTML apps. We do realize that the focus of the SysApps WG is currently on hosted use cases. However we believe it is possible to introduce a model which will be applicable to both modes, so that some of the [less entrenched] JS runtimes could choose to adopt it.

Another industry-specific consideration, related to permissions but not exactly equivalent to them, is business sensitivity of some automotive signals, and hence APIs that are used to access them. Thus the notion of “licensed” APIs, ones that are available based on some external arrangements to some apps but not others, is a concept widely accepted by automotive OEMs. This is already reflected in Availability APIs that are part of the automotive spec, currently under a public review. Our belief is that it should be possible to cover this notion under a comprehensive permissions system, though we are equally comfortable with leaving it within the automotive vertical

Here’s a link to a doc page that describes QNX’s current approach vis-à-vis permissions:

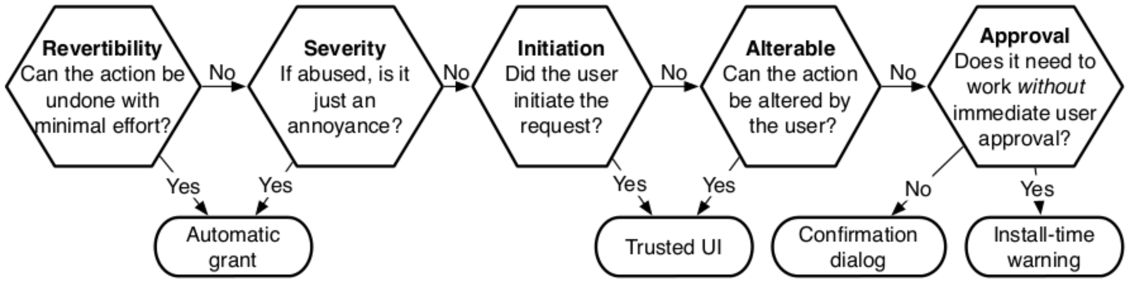
I believe all their apps are installed, not hosted

Dom: I guess the hope had been that these independently developed solutions to the permissions problem could converge in a standard approach. But that seems unlikely to succeed now.

B.1.4. Session 5: What lessons come out of academic studies

- Research project : Towards Comprehensible and Effective Permission Systems by Adrienne Porter Felt
- User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems, Roesner, et al.

Adrienne's hexagon diagram was warmly received (Figure 8.1 in her dissertation)


Revertibility
Can the action be undone with minimal effort?
No
Severity
If abused, is it just an annoyance?
No
Initiation
Did the user initiate the request?
No
Alterable
Can the action be altered by the user?
No
Approval
Does it need to work *without* immediate user approval?
Yes
Yes
Yes
Yes
No
Yes
Automatic grant
Trusted UI
Confirmation dialog
Install-time warning

This provides a decision graph for determining the most appropriate permission granting mechanism for a given capability: {automatic grant, trusted UI, confirmation dialog, install time warning}. The criteria include:

- Can the action be undone with minimal effort?
- If abused is the action just an annoyance?
- Did the user initiate the request?
- Can the action be altered by the user?
- Does it need to work without immediate user approval?

Two things to be added to the diagram:

1. The degree to which the capability is explainable to the user
2. How the capability will be used by the application

If understandable, is there any point in the app explaining what it wants it for (and how to trust that info).

Users want to know how a capability will be used instead of how it _can_ be used. But evil apps would lie about the intended use anyway. UAs should have a system for reporting apps that are abusing APIs/Capabilities. Basis for trust is is related to how much an app is reviewed by e.g. an app store. Is the browser the right tool for monitor the app behavior? Can be, but not specced by the W3C. Malware agents outside the scope of W3C.

We discussed the role of trusted UI as a natural means for users to grant permissions through user interaction, e.g. clicking on an icon of camera in other to take a photo. The browser disables the UI and greys it out if anything occludes its pixels. Likewise, the browser can apply a simple animation when the UI control is initially displayed as a means to signify that the control needs to be visible for a minimum period before it becomes active (analogous to speed bumps in roads that force drivers to slow down). The browser also needs to ensure user interaction with trusted UI controls can't be spoofed by applications.

Trusted UI may give developers some control over the rendering, e.g. the color palette or theme.

B.1.5. Session 6: Discussion of what considerations are important for the OWP

Tailoring user experience. Put a lot of justification in the app before asking for access to capabilities. Likewise, discovering when capability access has been revoked to show a suitable UI. Standards may be needed here. Naming of capabilities.

Result of this discussion may eventually be input to the WebApps WG work on manifest.

Granularity: different questions what is available to developer vs. what is presented to the user. Broad categories help for e.g. future proofing. What about vendor specific extensions?

Capabilities needed will vary depending on area (automotive, health, ….), and so may the permission model. But who should take care of future proofing? W3C CSS has worked out in this respect.

Should we develop specs or design patterns? Should it be in IG, CG or WG? Robin: does not really matter unless IPRs are in scope. Dom: Perhaps the Web and Mobile IG could host the work. But more important: who will have the time and interest to put in work?

Claes: can we produce anything normative? A lot is UI, difficult to be normative about that. Can anything normative be put (referenced to) from the UDP/TCP socket recommendation for example? There is a question if permission models should be specced, for other specs to reference.

MS: I'd definitely like to cover patterns and practices, let's see if we go beyond that (depends on if we have well defined use cases. We might spec how the developer declares how to spec the capabilities wanted (e.g. in a manifest). But people need to bring that question back to their organisations.

App store seem to be a central part, do we need app stores for the web? Broad question, outside the scope of this meeting perhaps.

Implications of different approaches have for API design:

- Some APIs only useful in certain contexts. Mislabeling controls to fool people.
- There should always be a failure callback (permissions may have been revoked).
- Need to check if the app has permissions or not.
- May not be a need to involve the user at all.
- Should be uniform across APIs.

Comes down to allow the developer to provide the best user exp.

Do we need patterns for persistence (once, rest of session, forever, ….)? Up to each UA, or something we should work on? Perhaps to default to per session.

B.2. THURSDAY, 4TH SEPTEMBER

We continue with the discussion of considerations we consider important for permission handling in the OWP.

GMandyam: I think in order to focus the discussion on trust, we should leave the discussion of app stores and the business processes they implement aside. Appsigning as an indication of trust has worked well, and the client does not have to have precise knowledge of the testing and validation that the application may have undergone in obtaining its signature in order for determining the level of trust in the application. Previous attempts at standardization of the app store and developer ecosystem (e.g. WAC, OMA) have not succeeded. The W3C should stay away from app store standardization if it wants to make progress in the area of application trust. Adrienne: if the browser uses heuristics to determine whether or not to grant permissions automatically or to ask the user or just to block some capabilities, this could be too surprising for users, and a potential pain for developers to figure out what they need to do to avoid the user permission prompts.

What kind of framework would be needed to support this?

Dom: pre-declaring required permissions might help, but we would need to look into the details, first.

John: we can’t avoid this problem if we are to give guidance to other groups. There are likely to be different implementations from different vendors, but we want to avoid the need for websites to have to apply different guidelines for different browsers.

Kenneth: web intents (Wonsuk: Ask to Kenneth whether does this should be replaced by Web components or not?) will complicate matters. (Claes: I think that Kenneth referred to web apps composed by several web components where the different components will require their permissions separately)

Adrienne: one case concerns trust across sites, the other is whether the browser trusts a site.

Steven: the browser wouldn’t trust apps when they make requests outside of their site

John: or more generally, controlled via CSP

Steven: issue when a developer makes a bad call when it comes to trusting 3rd party sites

John talks about a range of heuristics that browsers could employ. Fundamentally, we need some kind of delegated trust.

Adrienne: I agree. Otherwise iframes will need to ask for permissions separately, confusing the user.

GMandyam: If the user is actively managing website permissions using the browser chrome, then permission delagation needs to be managed accordingly. For instance, if the end user has denied permission for x.com to get access to the camera and then browses to y.com which tries to delagate permission back to x.com, then the browser needs to manage permissions approriately (e.g. blocking the delagated permission to x.com or prompting the user).

Stefan: sometimes you don’t trust the parent apps, but do trust the child app, currently no way to deal with this effectively, e.g. users can’t see the URL for the iframe.

Adrienne: users don’t understand apps which span websites.

John: in native world you can embed all kinds of stuff, e.g. maps, and you don’t care where it comes from. If we want web apps to offer similar user experience, we need to support some kind of trust delegation. In some cases it is okay for transitive trust to apply.

John asks how this could apply to the web?

Wendy: difference in scale, between installed apps (relatively few) and browsed-to websites. some questions around accountability and legal attribution

Wonsuk: likely need for contractual relationships between apps and nested apps around delegation of trust, and between app store vendor and app developers

Some discussion around level of curation involved

(FYI: http://www.w3.org/wiki/File:Manifest-usecases.png)

Wendy: you can imagine a liability regime which provides an incentive for the parent app developer to form contractual relationships or otherwise manage the trustworthiness of the sites it depends on for its services. I'm not sure we're there today.

Kenneth: today we see concerns around security and payments that lead to users being transferred from one site to another.

John talks about what it means to "install" (other trust gestures include "pinning", "bookmarking") hosted apps when you could review the trust and enable a more native like user experience as compared to sites you just visit (drive by sites)

Steven: upfront declarations have some bearing on this. It helps developers to think about what they are doing, and as well as helping with code review

Anssi: "pinning", "bookmarking", "installing" are trust gestures for keeping. Another type of indirect trust indicator is e.g. the frecency algorithm. The following is a brief extract from the document:

> *Frecency is a score given to each unique URI in Places, encompassing bookmarks, history and tags. This score is determined by the amount of revisitation, the type of those visits, how recent they were, and whether the URI was bookmarked or tagged. The word "frecency" itself is a combination of the words "frequency" and "recency."*

Steven: is there any value in W3C looking at using manifest for upfront declarations as part of the delegation model?

John: the browsers could be doing a lot behind the scenes, there could be some UI involved or none. My sense of the room is that some form of delegation is valuable.

Wendy: before we leave trust and delegation, consider attribution trails/ provenance: keeping (and perhaps exposing in some way) the chain of trust delegations.

Dave: when visiting arbitrary websites, the website itself could reference endorsements by third parties that are generally deemed trustworthy, but it is also interesting to consider active roles for 3rd party agents that monitor at what you are visiting and can flag good/bad sites, independently of the browser. This is already the case for the native world e.g. Lookout on android.

Dom: on native, the main app stores are operated by the same companies that are distributing the operating systems; trusting the OS is necessary, and extending that trust to the store, and implicitly to the apps these stores distribute makes some kind of sense; on the Web, the basic trust level is in the

User Agent; and the User-Agent plays a role in protecting user data, but also already goes a bit further than that by e.g. alerting the user on phishy web sites, or sites that are known to have malware via centrally maintained Web sites. Maybe there is more to be learned from that comparison, although we clearly don't want to the Web to become a closed/filtered ecosystem.

Adrienne: if there are many different ways for trust to accrue this could create complexity for developers as they seek to control the user experience

Kenneth: could it make sense to the user to "pin" a site to grant it additional permissions?

John: "pin" as a proxy for some gesture for indicating trust

Robin: "install" metaphor is easier for users to understand than pinning. That could be implied by installing as a gesture

Wendy: it makes sense to have an explicit user-gesture to grant increased permissions; should be something that users can easily understand

Adrienne: we're going to see websites yelling at users to pin them just as now they encourage users to install the site's native app

Robin: this is likely to be a short lived problem as the ecosystem matures (and 3rd party endorsements take off). Also consider buckets of permissions.

We move on to the broad category of questions around permission management

Dom: regarding indicators, we already know some specs (e.g. getUserMedia) set requirements on how indicators are expected to behave (if there are indicators); to me, this means we probably need to give guidance on whether that's a good a idea to set these requirements, and if it is, what requirements might look like.

There are some existing W3C specs that call out for indicators but don't specify the user experience for that.

Dave: Is there a need to inform apps when the user revokes a permission, e.g. to enable the app to dynamically adapt the user experience to match? i.e. by handling events rather than having to poll.

John: similarly, changes in system capabilities, e.g. caused by the user disconnecting something.

Kenneth: does native apps have a means to listen for such changes?

Steven: probably

John: it's not clear how important this is, but I don't want to rule it out just yet

Re privacy implications, we could drop a permission-denied signal and just return a failure.

(Related specs and Links - security/privacy considerations and best practices)

- http://www.w3.org/TR/geolocation-API/#security
- http://dev.w3.org/geo/api/spec-source.html#security
- http://www.w3.org/TR/web-intents/#privacy-considerations
- http://www.w3.org/TR/service-workers/#security-considerations
- http://www.w3.org/TR/html-media-capture/#security
- http://www.w3.org/TR/telephony/#security-and-privacy-considerations
- http://www.w3.org/TR/2010/NOTE-dap-privacy-reqs-20100629/
- http://www.w3.org/TR/app-privacy-bp/

Kenneth: there is a difference between background (app running with window/ tab in the background) or when the app windows is closed and the app is running as a system service (think Facebook Messages).

Giri: I agree, there is a difference between background as in service worker and visibility

Dom: most native platforms provide a number of APIs whose access is very limited (usually only to the OS maker or some device manufacturers); this provides isolation of the most dangerous or hard to explain permissions; is that something we want to extend to the Web, using e.g. Web Intents or an approach based on registerProtocolHandler? Should W3C deal with APIs to access these most limited APIs (as SysApps is currently chartered to do, e.g. the telephony API)?

Wendy: 3d party reviewers can assess the use of more granular permissions, give incentive to constrain access to what’s necessary

Lunch break and group photo, we resume at 13:30 French time

B.2.1. Session 7: Permissions-related API proposals

Adrienne introduced Mounir Lamouri's proposal for an API for developers to test whether a given permission had been granted, denied or would result in a prompt by the browser when the app tried to use the corresponding capability.

API Permissions articles/papers etc.:

- https://www.w3.org/wiki/Mobile/articles#API_Permissions Retired work items from the Device APIs WG:
- http://dev.w3.org/2009/dap/perms/FeaturePermissions.html
- http://www.w3.org/TR/api-perms/ Permissions API proposal posted to the public-webapps mailing list:
- https://docs.google.com/a/chromium.org/document/d/12xnZ_8P6rTpcGxBHiDPPCe7AUyCar-ndg8lh2KwMYkM/preview

- (a similar proposal, but with mixins: https://github.com/w3c/push-api/issues/3#issuecomment-43056068)

Steven: The four states I was referring to are:

1. Permission being sought and denied
2. Permission being sought and granted
3. Permission previously denied
4. Permission restricted

Giri: Are there still privacy considerations in exposing user preferences? {This seemed to be an issue the last time that DAP considered a permissions API} Could we reduce this concern by exposing “unavailable” rather than distinguishing permission not granted from unavailable? Also suggested that Mounir present this idea to PING to see if he can get some concrete feedback on any privacy concerns.

Vadim: Automotive, “availabiity” API, we have different reasons for non-availability, business, security, etc.

Adrienne: two different privacy considerations, fingerprinting

Wendy: Need to provide for the Tor browser use case: a browser that makes settings uniformly to provide an anonymity set

Robin: fingerprinting is probably a lost cause, but in the browser engine, you can detect sites doing too much investigation

Dave: look at ways the browser could protect you until the site is trusted; ways for third parties to assess apps at either install or run-time.

See also Nick Doty’s work with the Privacy Interest Group

Wendy: Can we bring in revocability? Useful in the API, e.g. to offer user “try this feature out for 30 min then automatically disable, unless you choose to keep it” without going out to the browser chrome.

Kenneth: can we use the names of the APIs as the names for permissions? Robin: yes, that was what we agreed yesterday

(Jonathan Jeon: we can reference from android case)

- http://developer.android.com/reference/android/Manifest.permission.html
- http://developer.android.com/reference/android/Manifest.permission_group.html

Steven: does the proposal include support for categories / groups of permissions? Adrienne: no, not at the moment.

Steven: still concerned about pre-flight request for permission. If dev has permission, should only be because they're using it. So why would you need the "has permission"?

John: in some cases it may be indeterminate whether a prompt will be needed, for instance, various factors may allow the browser to grant the permission without needing to ask the user.

Slightly favour exposing information that "this feature is not available *or • permission not granted" (it is not exposed to the web app whether this is due to a permission setting or unavailability of the device) rather than returning information"the feature is available *and* access to it has been granted". Don't promote harassing the user with "come on, please please grant this permission"

Steven: in principle, the user could disable a capability whilst the prompt for the permission is present. This means that you can't completely rely on the returned value.

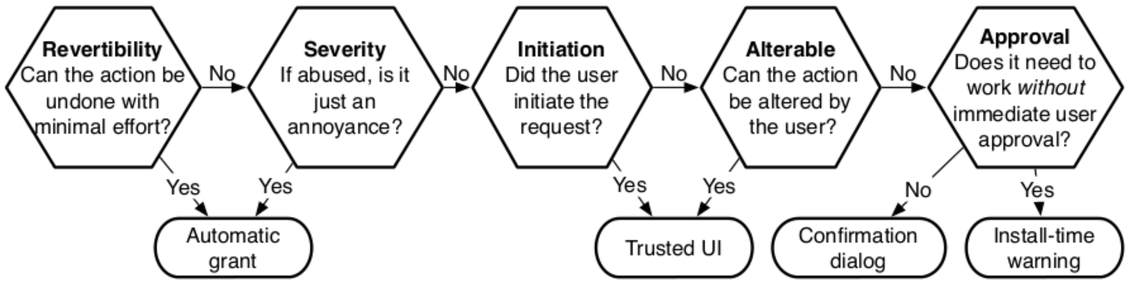
John: I am interested in the proposed API, but would like to see "prompt" replaced by a value that denotes that it is indeterminate whether a prompt would be invoked when attempting to use the associated capability.

We agree that we need names (strings) for permissions, e.g. for use in manifest and when dealing with requests for multiple permissions, but these names can be taken from the corresponding APIs, e.g. navigator.geolocation

B.2.2. Session 8: Plans for future work

(dom's link) web permissions requirements matrix

Adrienne's diagram of a decision graph for permission mechanisms:


Revertibility
Can the action be undone with minimal effort?
No
Severity
If abused, is it just an annoyance?
No
Initiation
Did the user initiate the request?
No
Alterable
Can the action be altered by the user?
No
Approval
Does it need to work *without* immediate user approval?
Yes
Yes
Automatic grant
Yes
Yes
Trusted UI
No
Confirmation dialog
Yes
Install-time warning

We agreed that the permission mechanisms need to be chosen according to the capabilities involved. Adrienne's diagram is very helpful for explaining some of the criteria involved. It would be valuable to provide guidelines on the principles to W3C working groups that are defining standards for APIs. This could be addressed by a W3C Community Group.

Trusted UI, that is embedded within web apps, is an promising area for further study as it allows user actions to implicitly grant permissions in a natural way without the need for the browser to prompt the user directly. The design of the corresponding UI requires a good understanding of the use cases. This too is something that could be addressed as part of a Community Group.

We agreed that trust delegation is essential to reducing the burden on users for understanding permissions. This would also permit finer grained permissions that make it easier for developers to only request the minimum capabilities they actually need. We can expect innovation by browser vendors for heuristic mechanisms for assessing apps and deciding when to ask the user and when to automatically grant or deny particular permissions. There needs to be some standards around how trusted third parties can endorse apps, especially for hosted apps.

W3C needs to define standards for permission handling for hosted apps on the Open Web Platform. Companies see a continuing need for packaged apps and we shouldn't rule these out, despite the lack of interoperability for packaged apps due to variations in APIs and packaging across vendors. We agree that declaring permissions in an app manifest can be useful for people reviewing apps, and as such facilitate trust in apps. A similar argument can be made for app developers to provide richer descriptions of what the app is using capabilities for. This information is aimed at reviewers and doesn't need to be presented to end users. There is a risk of developers trying to mislead reviewers, but this can be countered through having multiple independent reviews and through innovations in reviewing practices and heuristic tools for accessing apps.

We also liked the idea of enabling app developers to adapt the user experience according to what permissions have been granted and are available as capabilities. This could make it a little easier to finger print devices, but there are already many ways in which to do so, and rather than worrying about adding extra bits of finger printing information, it is more appropriate to consider ways for browsers to apply heuristics to flag up apps that appear to be using finger printing. Reviewers could then decide whether the app is behaving reasonably given its stated purpose, or whether it is a rogue app.

There is further work to be done on the details of the permission models, for example, whether developers have access to {granted, denied, prompt} for permissions, or whether the permissions APIs should not distinguish between a permission being denied, and the corresponding capability not being available. The persistence model for permissions is expected to be an area where browser innovation around trust is likely to make it hard to standardize in a prescriptive way.

Areas of Rough Consensus on Work Items (and which groups)

- Handling of trust delegation to iframes, web components (Adrienne) (WebAppSec? (CSP) HTML? (doing sandbox already))
- Extension of manifest to declare permissions needed/features used (WebApps WG)
    - http://w3c.github.io/manifest/
- Permissions API proposal (WebApps WG (proposed there)? SysApps? DAP (could already be in charter (Dom)? WebAppSec ? PING should review)
    - DAP Charter (http://www.w3.org/2011/07/DeviceAPICharter) has the following work item in scope "An API for requesting and managing user permissions to use device features"
- Review of APIs wrt how they handle permissions (Best Practices) (Mobile Web IG? TAG? DAP? WebApp? (important: find somebody to do the work (Adrienne interested) [Dom interested to help if not alone]))
    - Dom's review of existing APIs permissions in Mobile Web IG (but really Dom only - Adrienne/Chrome have content to contribute)
    - Document existing practices for permission handling http://www.w3.org/TR/app-privacy-bp/
        - http://lists.w3.org/Archives/Public/public-web-mobile/2014Jan/0001.html
- Best practices for browser vendors (Microsoft, Google, Qualcomm interested - CG?)
    - not a normative document
    - Dom's review of existing APIs permissions also relevant here
    - enterprise rules & policy
    - Adrienne's work (flowchart)
    - helpful for partcipants to agree on best practices
    - not necessarily read
    - Trusted UIs, gadgets rather than prompts (Dave)
        - researchy
- (Packaged apps not ruled out)

Areas where we are still some way apart but agree to work on closing the gap

- (how about permission model ?)
- An interoperable way for indicating trust, e.g. endorsements for hosted web apps (premature to determine group? put into CG above)
    - Use cases to document what we want to achieve wrt trust delegation
    - Permissions on sensitive APIs (gemalto)
        - standard way to represent endorsement as "trustworthy" by some party (signature)
        - like certificate authorities
        - (some debate on whether this should be done)
        - or signed webapp "packages". not a CA
- (what considerations from mission critical case - web and automotive ?)

B.2.3. Next Steps

The W3C staff are planning on a blog post to draw attention to the meeting at its conclusions. A session will be held at TPAC to further encourage discussions on next steps including the launch of the Community Group.

B.2.4. Addendum

Jonathan Jeon made the following suggestions after the meeting:

> *I didn't propose the issues during this meeting, but I think we need to consider as below issues for future work.*
>
> - *Wide scope permission issue on web app store ecosystem:*
>
>   *As we know well, Web app store will be a key component on web app ecosystem. So if we want to make an open web app ecosystem, we need to make the way for Open Web App Store Federation. Web app store federation is also related with app permission issue. The Web Application Store Community Group has prepared some Requirements and Use Cases for Open Web Application Store.*
>
> - *Permission & APIs for system setting:*
>
>   *If we will make the permission APIs, it would be good to consider the API for high level system setting. High level setting API can be provide a simple interface to manage the global system-level device preferences and setting. Here is a proposal for a System Settings API.*

Our thanks to Gemalto for hosting this meeting and to all those who participated.

C. MANIFEST FOR WEB APPS AND BOOKMARKS

W3C First Public Working Draft 17 December 2013

This version:
http://www.w3.org/TR/2013/WD-appmanifest-20131217/

Latest published version:
http://www.w3.org/TR/appmanifest/

Latest Editor's draft:
http://w3c.github.io/manifest/ (commit history)

Editors:
Marcos Caceres, Mozilla Corporation

Anssi Kostiainen, Intel Corporation

Kenneth Rohde Christiansen, Intel Corporation

Feedback?
We are on GitHub

File a bug

C.1. ABSTRACT

This is a work in progress! For the latest updates from the Web Applications (WebApps) Working Group possibly including important bug fixes, please look at the draft on GitHub.

This specification defines a *manifest*, which provides developers with a centralized place to put metadata about a web application. This includes, amongst other things, the ability to specify the name of the web application, links to icons, as well as the preferred URL at which the web application should open when it is launched by the user.

With this metadata, user agents can provide, for example, enhanced bookmarking capabilities such as being able to add a web application to the homescreen of a device - as well as the various icons needed to effectively integrate with an OS's task switcher and system preferences. The specification also defines an API to enable bookmarking from within a document, as well as a means to check if an application is running in a special mode called "standalone".

This specification also defines the `manifest` link type, which provides a declarative means for a web document to be associated with a manifest.

C.2. STATUS OF THIS DOCUMENT

This is a work in progress! This specification is for review and not for implementation! For the latest updates, including important bug fixes, please look at the draft on GitHub instead.

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

Implementors should be aware that this specification is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation phase should join the aforementioned mailing lists and take part in the discussions.

This document was published by the Web Applications (WebApps) Working Group as a First Public Working Draft. This document is intended to become a W3C Recommendation. If you wish to make comments regarding this document, please send them to public-webapps@w3.org (subscribe, archives). All comments are welcome. You can also File a bug.

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

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

C.3. TABLE OF CONTENTS

C.4. 1. USAGE EXAMPLES

This section shows how the expected usage of the various features provided by this specification.

C.4.1. 1.1 Example manifest

This section is non-normative.

The following shows a typical manifest.

Example 1: typical manifest

```
{
  "name": "Example",
  "url": "/start.html",
  "mode": "standalone",
  "icons": [{
      "src": "icon/lowres",
      "density": "1",
      "width": "64",
      "type": "image/webp"
    }, {
      "src": "icon/hd",
      "density": "2",
      "width": "64"
  }]
}
```

C.4.2. 1.2 Example of linking to manifest

Example of using a `link` element to associate a website with a manifest. The example also shows how HTML fallbacks, such as "application-name", can be used to support legacy user agents that don't implement this specification.

Example 2: linking to a manifest

```
<!doctype html>
<html>
<head>
<title>The Best News - international</title>

<!-- link to bookmark metadata -->
<link rel="manifest" href="add_to_homescreen.json">

<!-- fallback metadata for legacy browsers -->
<meta name="application-name" content="Best News!">
<link rel="shortcut icon" src="favicon.ico">
</head>
...
```

C.4.3. 1.3 Example of using the API

Check if the application is running in standalone mode.

Example 3: detecting standalone mode

```
<script>
if("standalone" in navigator && navigator.standalone){
   //Do standalone specific stuff...
   document.documentElement.classList.add("standalone");
}
</script>
```

C.4.4. 1.4 Request adding to homescreen

Let the user bookmark the application through clicking a button.

Example 4: requesting to bookmark a web app

```
<script>
var installButton = document.querySelector("#install");

//Bookmarking only works if initiated by user interaction
installButton.addEventListener("click", function(e){
   navigator.requestBookmark("path/to/bookmark.json")
   .then(thankUser, stopBuggingUser);
});
```

```
</script>

<button id="install">
  Bookmark <b>Awesome App!</b>
</button>
```

C.5. 2. USE CASES AND REQUIREMENTS

This document attempts to address the *Use Cases and Requirements for Installable Web Apps*.

C.6. 3. MANIFEST

A *manifest* is a [*JSON*] document that consists of a top-level object that can contain zero or more members, some of which contain other objects. Each of the members, as well as how their values are processes, are defined below.

Algorithms in this specification use the conventions described in [*ECMASCRIPT*], such as the use of steps and sub-steps, and so on. The *parseFloat* method, *ToString*, *HasOwnProperty*, *[[GetOwnProperty]]*, and *ToBoolean* abstract operations, and the *Type*(*x*) notation referenced in this section are defined in [*ECMASCRIPT*]. Processing also relies on various algorithms defined in [*HTML*], [*FETCH*], and [*URL*].

As the manifest format is a [*JSON*] document, this specification relies on the types defined in [*JSON*] specification: namely *object*, *array*, *number*, and *string*. Strict type checking is not enforced by this specification. Instead, each member's definition specifies the steps required to process a particular member.

When an algorithm asks the user agent to *issue a developer warning*, the user agent *MAY* report the conformance violation to the developer in a user-agent-specific manner (e.g., show the problem in the error console), or *MAY* ignore the error and do nothing

In the algorithms, to *ignore* means that the user agent *MUST* act as if the developer had not declared the particular member in the manifest document.

C.6.1. 3.1 `name` member

The *`name`* is a string that represents the name of the application as is usually displayed to the user (e.g., amongst a list of other applications, or as a label underneath an icon).

The *steps for processing the name of an application* is given by the following algorithm. The algorithm takes a *manifest* as an argument. It returns either `undefined`, a string, or an error.

1. If HasOwnProperty(*manifest*, "name") returns `false`, then return `undefined`.
2. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *manifest* with argument "name".
3. If Type(*value*) is not "string", return an error.
4. Strip leading and trailing whitespace from *result*.
5. Return the *result*.

C.6.2. 3.2 `dont-share-cookies-and-stuff` member

Issue 1

Yes, this needs to be renamed. Possible candidates: "isolated" or "isolated-security-origin", "independent"

The `dont-share-cookies-and-stuff` member is a boolean that allows a developer to request that the user agent treat this web application as independent from the one in the web browser. Effectively, this means that cookies, storage, and permissions are not shared with the origin from which the application was bookmarked. The value of this member is only applicable to applications whose mode of operation is standalone.

The *steps for processing the `dont-share-cookies-and-stuff` member* is given by the following algorithm:

1. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *manifest* with argument "`dont-share-cookies-and-stuff`".
2. Let *result* be the result of calling ToBoolean(*value*).
3. Return the *result*.

C.6.3. 3.3 `url` member

The *url* member is the URL that is loaded when the application is launched. When it's missing from the manifest, the UA loads the URL from the manifest was fetched.

The *steps for processing the `url` member* are given by the following algorithm. The algorithm returns a URL.

1. If HasOwnProperty(*manifest*, "url") returns `false`, then return `undefined`.
2. Let *value* be the result of calling the [[GetOwnProperty]] internal method of the *manifest* with argument "`url`".
3. Let *manifest URL* be the URL from which the *manifest* was fetched from.
4. If *value* is `undefined` or Type(*value*) is not "string":
    1. Let *result* be *manifest URL*.
5. Otherwise:
    1. Parse *value*, using *manifest URL* as the base URL, and let *result* be the result.

6. Return *result*.

C.6.4. 3.4 `icons` member

The *icons* is a list of icon objects that represents a set of icons that the application can make use.

The *steps for processing the `icons` member* are given by the following algorithm. The algorithm returns a list of icons, which can be empty, or `undefined`.

1. If HasOwnProperty(*manifest*, *key*) returns `false`, then return `undefined`.
2. Let *icons* be an empty list.
3. Let *manifest URL* be the URL from which the *manifest* was fetched from.
4. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *manifest* with argument "`icons`".
5. If *value* is an array, then for each *potential icon* in the array:
    1. Let *src* be the result of running the steps for processing the `src` member of an icon. If the result is an error, stop processing this *potential icon* and move to the next *potential icon* in value, if any.
    2. Let *type* be the result of running the steps for processing the `type` member of an icon.
    3. If *type* is not a valid MIME type or the value of type is not a supported icon format, then stop processing this *potential icon* and move to the next *potential icon*, if any.
    4. Let *width* be the result of running the steps for processing the dimensions of an icon with "width" as the argument.
    5. Let *height* be the result of running the steps for processing the dimensions of an icon with "height" as the argument.
    6. Otherwise, let *icon* be an object.
6. return *icons*.

C.6.5. 3.5 `orientation` member

Issue 2

This is issue 74 on GitHub. We are looking for feedback!

Orientation of an application is dependent on the media features of the display. For example an application might need to be launched in landscape on phones (in order to have sufficient display width), but prefer to be in portrait on tablets. (see Orientation section in the use cases document).

When analyzing applications across various runtimes, we've found evidence that such applications are common (e.g., basically any application on the iPhone that has an iPad counterpart will be designed to constrain to a particular orientation based on the device being used: LinkedIn, Flipboard, GoodReads, etc. will all go from portrait-primary on the iPhone to allowing "any" orientation on the iPad. A more extreme example is BBC iPlayer - which supports portrait-primary on the iPhone, but both landscape orientations on iPad. The same can be seen on Android devices. Unlike native apps, Web Apps should not target devices/OS's - they have to be device neutral.

In order to address the use cases, we currently have two proposals.

Option 1: Provide a list of orientation sets in the manifest. The user agent uses the first one with a matching media query. The order in which the orientations are listed by a developer does not imply a preference for setting the orientation - it is always left up to the user agent to pick the best orientation given, for example, how the user is holding the device. In the example below, no orientation is given for widths of 721px or above, so the default is used: allowing all orientations supported by the device.

```
{
    "orientations": [{
        "media": "max-width: 320px",
        "supported": ["portrait-primary", "landscape"]
    }, {
        "media": "max-width: 720px",
        "supported": ["landscape"]
    }]
}
```

In this example:

- a device with a screen width of 320px or below would launch either "portrait-primary" or "landscape" with the abilty to be "flipped" depending on how the user is holding the device (and OS permitting).
- A device with a screen width of 321px through 720px would be launched in landscape (leaving it up to the UA to pick either landscape-primary or landscape-secondary, while allowing "flippability"),
- A device with a screen width of 721px and above would start in any orientation chosen by the UA (ideally, one that matches how the user is holding the device).

Option 2: The second proposal is to remove orientation from the manifest and use CSS @viewport instead [*css-device-adapt*]. This would mean::

```
<head>
<style>
  /*set it by default to portrait primary for small screens */
  @media (max-width: 320px) {
      @viewport {
          orientation: portrait-primary, landscape;
      }
  }

  /*Tablet, switch to landscape only*/
  @media (max-width: 720px) {
      @viewport {
          orientation: landscape;
      }
  }

  /*
    similarly on screens with a width of 721px or more,
    all orientations are allowed
   */
</style>
</head>
```

Problem with using `@viewport` at the moment is that the specification is progressing a bit slowly and no one has implemented the "orientation" descriptor. It also lacks definitions for "-primary" and "-secondary" contraints, which are important for various applications, and doesn't currently allow providing multiple allowed orientations - hopefully the CSS Device Adaption spec can align with the Screen Orientation spec.

C.6.6. 3.6 `mode` member

The *mode* member represents that mode of operation in which the web application will be launched. When the developer omits the value, the user agent assumes the value "bookmark".

The *mode of operation* in which a web application can be launched include:

standalone
: Once launched, the Web application appears indistinguishable from a native application by allowing the OS to treat it as equivalent to a native application.

bookmark
: When launched, the user agent open the URL is normal (e.g., opens a new tab, or a new window, or whatever it commonly does when instructed to open a bookmark).

Standalone mode can also be displayed in the following view modes:

A *valid application mode* is one that conforms to the following [*ABNF*]:

```
  mode = "bookmark" / "standalone" ["-" presentation]
  presentation = "fullscreen"
```

The *steps for processing the `mode` member of an icon* are given by the following algorithm. The algorithm returns a string.

1. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *manifest* passing "`mode`" as the argument.
2. Let *result* be "bookmark".
3. Strip leading and trailing whitespace from *value*.

4. If *value* is a valid application mode, set *result* to *value*.
5. Return *result*.

C.6.7. 3.7 Processing the manifest

The *steps for fetching a manifest* are given by the following algorithm. It takes a *url* URL as an argument. It returns either a *response* (which may be in error).

1. Let *response* be the result of fetching the manifest from *url*.
2. Return *response*.

The *steps to processing a manifest* are given by the following algorithm. The algorithm takes a *text* string as an argument.

1. Let *manifest* be the result of invoking the parse function of the JSON object defined in [*ECMASCRIPT*] with *text* as its only argument. If parsing throws an error, return the error and terminate this algorithm.
2. Let *name* be the result of running the steps for processing the `name` member. If the returned value is `undefined` or an error:
    1. Set *name* to be an implementation specific string that can serve as a suitable name for the web application. For example, if the manifest was obtained from a `link` element, and the `document` has a `meta` element whose `name` attribute matches "`application-name`", then the user agent could use the value of that `meta` element's `content` attribute as a fallback. Otherwise, then the document's `title` can be used. Alternatively, the user could be prompted to provide a custom name using as a place holder.
3. Let *mode* be the result of running the steps for processing the `mode` member. If processing returns an error, report the error, and set *mode* to "`bookmark`".
4. Let *orientation* be the result of running the steps for processing the `orientation` member.
5. Let *url* be the result of running the steps for processing the `url` member.
6. Let *icons* be the result of running the steps for processing the `icons` member.
7. Optionally, ignore all other members.
8. Return response.

C.6.8. 3.8 Linking to a manifest

The `manifest` keyword can be used with a [*HTML*] link element. This keyword creates an external resource link.

Link type	**Effect on...**		**Brief description**
	`link`	**`a` and `area`**	
`manifest`	External Resource	not allowed	Imports or links to a manifest.

The default media type for resources associated with the `manifest` link type is `application/manifest+json`.

In cases where more than one `link` element with a `manifest` link type appears in a `document`, the user agent *MUST* use the first inserted link element and ignore all subsequent `link` elements with a `manifest` link type (even if the first element was in error).

The appropriate time to fetch the manifest is when the external resource link is created or when its element is inserted into a document, whichever happens last. However, a user agent *MAY* opt to delay fetching a manifest until after the document and its other resources have loaded (i.e., to not delay the availability of content and scripts required by the `document`).

Certain error conditions can result in a manifest being treated as an invalid manifest. An *invalid manifest* is one that is deemed to be non-conforming in such a way that it would not be possible for the user agent to continue processing (e.g., it can't be parsed by the JSON parser because of a syntax error, it could be fetched from the network). In such a case, issue a developer warning. In either case, when a step results in an invalid manifest the user agent *MUST* abort whatever step or sub-step caused the condition.

To fetch a manifest, as user agent *MUST*:

1. If the `link` element lacks a href attribute, abort this algorithm.
2. Run the steps for fetching a manifest, with the value of `href` attribute as the url, and let *response* be the response.
3. If response's type is "error", treat this as an invalid manifest.
4. Otherwise, let *manifest* be the result of running the steps for processing a manifest.
5. If *manifest* results in an error, treat this as an invalid manifest.
6. Otherwise, in a user agent specific manner, and when the end-user so desires, provide a means for the end-user to view the relevant contents of the *manifest*; and provide a means for the end-user to create a bookmark based on that information.

C.6.9. 3.9 Proprietary extensions to the manifest

This section is non-normative.

Although proprietary extensions are undesirable, they can't realistically be avoided. As such, the *RECOMMENDED* way to add proprietary extension is to use a vendor prefix.

This following is an example of two hypothetical vendor extensions.

Example 5: vendor extensions

```
{
  ...
  "webkit-fancyfeature": "some/url/img",
  "moz-awesome-thing": { ... }
  ...
}
```

C.7. 4. LAUNCHING A STANDALONE WEB APPLICATION

When an application is launched:

1. If the web application has a required orientation, run the steps for selecting the orientation.

C.8. 5. ICON OBJECT AND ITS MEMBERS

Each *icon object* represents an icon for an application suitable to use at some dimensions and screen density.

C.8.1. 5.1 `density` member

The *density* member of an icon is the device pixel density for which this icon was designed. The device pixel density is expressed as the number of dots per 'px' unit (equivalent to a dppx as defined in [*css3-values*]). The value is a positive number greater than 0. If the developer ommits the value, the user agent assumes the value `1`.

The *steps for processing a density of an icon* are given by the following algorithm. The algorithm thanks an *icon* object as an argument and returns a positive number.

1. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *icon* passing "`density`" as the argument.
2. Let *result* be the result of parseFloat(*value*);
3. If *result* is `NaN`, +0, −0, +∞, or −∞, or less than 0, return 1.
4. Return *result*.

C.8.2. 5.2 `width` and `height` members

The *width* and *height* members represent the natural width of the icon in pixels. Their corresponding value is a positive number greater than 0.

The *steps for processing a dimension of an icon* are given by the following algorithm. The algorithm takes an *icon* and a *key* ('width" or "height") as an argument. The algorithm returns a positive number, `undefined`, or an error.

1. If HasOwnProperty(*icon*, *key*) returns `false`, then return `undefined`.

2. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *icon* passing *key* as the argument.
3. Let *result* be the result of parseFloat(*value*);
4. If *result* is `NaN`, +0, −0, +∞, or −∞, or less than 0, return *undefined*.
5. Return *result*.

C.8.3. 5.3 `src` member

The *src* member of an icon is a URL from which the icon can be fetched.

The *steps for processing the `src` member of an icon* are given by the following algorithm. The algorithm takes a *icon* object as an argument and returns a URL or `undefined`.

1. If HasOwnProperty(*icon*, "src") returns `false`, then return `undefined`.
2. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *icon* passing "`src`" as the argument.
3. Let *manifest URL* be the URL from which the *manifest* was fetched from.
4. Parse *value*, using *manifest URL* as the base URL, and let *result* be the result.
5. return *result*.

C.8.4. 5.4 `type` member

The *type* member of an icon is a hint as to the media type of the icon. The purpose of this member is to allow a user agent can ignore icons of media types it does not support.

The *steps for processing the `type` member of an icon* are given by the following algorithm. The algorithm takes an *icon* object as an argument, and returns either string or `undefined`.

1. If HasOwnProperty(*manifest*,"type") returns `false`, then return `undefined`.
2. Let *value* be the result of calling the [[GetOwnProperty]] internal method of *potential icon* passing "`type`" as the argument.
3. If type(*potential src*) is not "string", return an error.
4. Strip leading and trailing whitespace from *result*.
5. return *result*.

C.9. 6. EXTENSIONS TO THE `NAVIGATOR` OBJECT

```
partial interface Navigator {
    readonly    attribute Boolean standalone;
    Promise requestBookmark (optional DOMString url);
};
```

C.9.1. 6.1 Attributes

`standalone` of type Boolean, readonly

The standalone attribute provides a means for a developer to check if the application is running in standalone mode. When getting, the user agent *MUST* returns `true` if the application is running as standalone. Return `false` otherwise.

C.9.2. 6.2 Methods

`requestBookmark`

When invoked, the user agent *MUST* run the steps to request to add bookmark.

Parameter	Type	Nullable	Optional	Description
url	`DOMString`	✘	✔	

Return type: `Promise`

The *steps to request to add bookmark* are given by the following algorithm. The algorithm runs asynchronously and returns a Promise.

1. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
2. Return *promise* and run the remaining steps asynchronously.
3. If this method was not invoked as a result of explicit user action, then:
    1. Let *error* be a new `DOMException` whose name is "`SecurityError`".
    2. Run resolver's internal reject algorithm with *error* as value and terminate this algorithm.
4. If the document's mode of operation is "standalone":
    1. Let *error* be a new `DOMException` whose name is "`NotSupported`".
    2. Run resolver's internal reject algorithm with *error* as value and terminate this algorithm.
5. Run the steps for fetching a manifest. If the *response*'s type is "error":
    1. Let exception name be "`NetworkError`".
    2. If the *response*'s termination reason is end user abort, set *exception name* to "`AbortError`".
    3. If the *response*'s termination reason is timeout, set *exception name* to "`TimeoutError`".
    4. Let *error* be a new `DOMException` whose name is *exception name*.
    5. Run *resolver*'s internal reject algorithm with *error* as value and terminate this algorithm.
6. Let *boomark info* be result of the steps for processing a manifest, with *response*'s body as the argument.
7. If *boomark info* is an error:
    1. Run *resolver*'s internal reject algorithm with *error* as value and terminate this algorithm.

8. Present *bookmark info* to the end user. If the end-user rejects the request to add the bookmark:
    1. Let *error* be a new `DOMException` whose name is "`AbortError`".
    2. Run *resolver*'s internal reject algorithm with *error* as value and terminate this algorithm.
9. Run *resolver*'s internal fulfill algorithm with *undefined* as value.

C.10. 7. MEDIA TYPE REREGISTRATION

This section contains the required text for MIME media type registration with IANA.

The *media type for a manifests* is `application/manifest+json`.

If the protocol over which the manifest is transferred supports the [*MIME-TYPES*] specification (e.g. HTTP), it is *RECOMMENDED* that the manifest be labeled with the media type for a manifests.

Type name:
application

Subtype name:
manifest+json

Required parameters:
N/A

Optional parameters:
N/A

Encoding considerations:
Same as for application/json

Security considerations:

As the manifest format is JSON and will commonly be encoded using [*UNICODE*], the security considerations described in [*JSON*] and [*UNICODE-SECURITY*] apply. In addition, implementors need to impose their own implementation-specific limits on the values of otherwise unconstrained member types, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

Web applications will generally contain ECMAScript, HTML, CSS files, and other media, which are executed in a sand-boxed environment. As such, implementors need to be aware of the security implications for the types they support. Specifically, implementors need to consider the security implications outlined in the [*CSS-MIME*] specification, the [*ECMAScript-MIME*] specification, and the [*HTML*] specification.

As web applications can contain content that is able to simultaneously interact with the local device and a remote host, implementors need to consider the privacy implications resulting from exposing private information to a remote host. Mitigation and in-depth defensive measures are an implementation responsibility and not prescribed by this specification. However, in designing these measures, implementors are advised to enable user awareness of information sharing, and to provide easy access to interfaces that enable revocation of permissions.

As this specification allows for the declaration of URLs within certain members of a manifest, implementors need to consider the security considerations discussed in the [*URL*] specification. Implementations intending to display IRIs and IDNA addresses found in the manifest are strongly encouraged to follow the security advice given in [*UNICODE-SECURITY*].

Applications that use this media type:
Web browsers

Additional information:
Magic number(s):
N/A

File extension(s):
.json, .manifest

Macintosh file type code(s):
TEXT

Person & email address to contact for further information:
The Web Applications (WebApps) Working Group can be contacted at public-webapps@w3.org.

Intended usage:
COMMON

Restrictions on usage:
none

Author:
W3C's Web Applications (WebApps) Working Group.

Change controller:
W3C.

C.11. 8. CONFORMANCE

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

The key words *MUST*, *MUST NOT*, *REQUIRED*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY*, and *OPTIONAL* in this specification are to be interpreted as described in [*RFC2119*].

There is only one class of product that can claim conformance to this specification: a *user agent*.

C.12. A. ACKNOWLEDGMENTS

This document reuses text from the WHATWG [*HTML*] specification as permitted by the license of that specification. The [*HTML*] specification is edited by Ian Hickson.

C.13. B. REFERENCES

C.13.1. B.1 Normative references

[ABNF]
D. Crocker; P. Overell. *Augmented BNF for Syntax Specifications: ABNF*. January 2008. STD. URL: http://www.ietf.org/rfc/rfc5234.txt

[CSS-MIME]
H. Lie; B. Bos; C. Lilley. *The text/css Media Type*. 1 March 1998. Informational. URL: http://www.ietf.org/rfc/rfc2318.txt

[ECMASCRIPT]
ECMA-262 ECMAScript Language Specification, Edition 6. Draft. URL: http://people.mozilla.org/~jorendorff/es6-draft.html

[ECMAScript-MIME]
B. Hoehrmann. *Scripting Media Types*. 4 January 2006. Informational. URL: http://tools.ietf.org/html/rfc4329

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

[HTML]
Ian Hickson. *HTML*. Living Standard. URL: http://www.whatwg.org/specs/web-apps/current-work/

[JSON]
D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON) (RFC 4627)*. July 2006. RFC. URL: http://www.ietf.org/rfc/rfc4627.txt

[MIME-TYPES]
N. Freed; N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types (RFC 2046)*. November 1996. RFC. URL: http://www.ietf.org/rfc/rfc2046.txt

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt

[UNICODE]
The Unicode Standard. URL: http://www.unicode.org/versions/latest/

[UNICODE-SECURITY]
Mark Davis; Michel Suignard. *Unicode Security Considerations*. URL: http://www.unicode.org/reports/tr36/

[URL]
Anne van Kesteren. *URL Standard*. Living Standard. URL: http://url.spec.whatwg.org/

[css3-values]
Håkon Wium Lie; Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 3*. 30 July 2013. W3C Candidate Recommendation. URL: http://www.w3.org/TR/css3-values/

C.13.2. B.2 Informative references

[css-device-adapt]
Rune Lillesveen. *CSS Device Adaptation*. 15 September 2011. W3C Working Draft. URL: http://www.w3.org/TR/css-device-adapt/

D. THE APP: URL SCHEME

W3C Last Call Working Draft 29 May 2014

This version:
http://www.w3.org/TR/2014/WD-app-uri-20140529/

Latest version:
http://www.w3.org/TR/app-uri/

Editor's draft:
http://app-uri.sysapps.org/

Previous version:
http://www.w3.org/TR/2013/WD-app-uri-20130516/

Feedback:
public-sysapps@w3.org/ with subject line “[appuri] … message topic …” (archives)

Editor:
Marcos Caceres, Mozilla Corporation

Repository:
We are on Github.

File a bug.

Commit history.

D.1. ABSTRACT

This specification defines the app: URL scheme.

The app: URL scheme can be used by packaged applications to obtain resources that are inside a container. These resources can then be used with web platform features that accept URLs.

D.2. STATUS OF THIS DOCUMENT

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

This document was published by the System Applications Working Group as a Last Call Working Draft. If you wish to make comments regarding this document, please send them to public-sysapps@w3.org (subscribe, archives). All comments are welcome.

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

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

This specification is a **Last Call Working Draft**. All persons are encouraged to review this document and **send comments to the public-sysapps mailing list** as described above. The **deadline for comments** is **24 June 2014**.

D.3. TABLE OF CONTENTS

D.4. 1. APP: URL

An *app: URL* is a [*URL*] that can be used by a packaged application to address resources within its container (e.g., a .zip file).

D.5. 2. INSTANCE IDENTIFIER

The *instance identifier* is a string that uniquely identifies an instance of a packaged application.

When the instance identifier is not provided by the developer, a user agent *MUST* synthesize one. The structure and length of identifier represented by the authority is application specific, but it *MUST* be suitable to use as a `Document`'s origin. See also privacy and security considerations.

D.5.1. 2.1 Privacy Considerations

This section is non-normative.

Using unique identifiers (e.g., a UUID) as an instance identifier can be exploited by an adversary as a digital finger print. This can allow a developer to, for example, restore cookies even if the user has cleared cookies from a user agent. As such, if the user agent relies on unique identifiers as the host component, then it should provide end-users with a means of regenerating the authority component. For instance, A user agent can the regenerate the instance identifier when the user clears the user agent's private data.

D.6. 3. FETCHING A RESOURCE FROM A CONTAINER

This section is non-normative.

To *fetch a resource using the app: URL* using a *request* request:

1. If *request*'s method is not `` `GET` ``, or if *origin* does not match the instance identifier for this application, return a network error.
2. Let *path* be the path of *URL*.
3. Let *response* be a response.
4. If attempting to access the resource at *path* results in an error (e.g., not found, the file is corrupt, locked, etc.), return a network error and terminate this algorithm.
5. If *request* includes a `` `Range` `` header:
    1. If the value of the `` `Range` `` header is a valid byte range [*HTTP11*]:
        1. Set *response* body to be the data from *path* at the start and end of `` `Range` ``.
        2. Set *response* status to 206.
6. Otherwise, set the *response* body to be the data at *path*.
7. Set the *response* headers:
    1. `` `Content-Length` ``, computed as per [*HTTP11*].
    2. `` `Content-Type` ``, the MIME type of the resource at *path* computed as per [*SNIFF*].
8. Return *response*.

D.6.1. 3.1 Security considerations

When fetching data from an app: URL, a user agent needs to make sure that only files that were in the container can be accessed (i.e., those files should be sand-boxed). User agents need to watch out for symbolic links (or similar) inside a container, which can attempt to trick the user agent into accessing files that are on other parts of the file system.

D.7. 4. CONFORMANCE

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

The key words *MUST*, *MUST NOT*, *REQUIRED*, *SHOULD*, *SHOULD NOT*, *RECOMMENDED*, *MAY*, and *OPTIONAL* in this specification are to be interpreted as described in [*RFC2119*].

There is one class of product that can claim conformance to this specification: a user agent.

A *user agent* is an implementation of this specification.

D.8. A. USE CASES

For developers, this specification attempts to address the following use cases:

1. Provide a URL scheme that is compatible with the features and security model of [*HTML*], so that applications that are packaged can make full use of web platform's capabilities. For example, the app: URL scheme should be usable as a document's address so that it can serve as its origin, and it should be compatible with APIs like [*XHR*].
2. Provide fetching model that constrains retrieval of files to a specific container.
3. Provide an addressing scheme that is easy to work with - and that developers are already accustomed to working with. The developer should not be bothered as to whether they are using `http://` or `app://` - the Web's capabilities and APIs need to just work!
4. Support the ability to playback audio and video files within a packaged web application, including the ability to seek without needing to load the full resource.

D.9. B. EXAMPLES

The following example shows [HTML]'s `window.location` using then app: URL.

Example 1

```
<!doctype html>
<script>
//Example using HTML's Location object
var loc =  window.location;
console.log(loc.protocol === "app:"); //true
console.log(loc.host === "com.foo.bar"); //true
console.log(loc.href === "app://com.foo.bar/index.html"); //true
console.log(loc.origin === "app://com.foo.bar"); //true
console.log(loc.pathname === "/index.html"); //true
console.log(loc.hash === "#example"); //true
console.log(loc.port === ""); //true
</script>
```

This example shows an app: URL being resolved in [*HTML*].

Example 2

```
var img = document.createElement("img");

//the following setter triggers HTML's resolve algorithm
img.src = "example.gif";

//and the expected output:
console.log(img.src === "app://c13c6f30/example.gif") //true

//Append the image to the document
document.body.appendChild(img);
</script>
```

This example shows a resource within a packaged application being retrieved over [*XHR*].

Example 3

```
function process() {
  // process the resulting data
}

var xhr = new XMLHttpRequest();
xhr.onload = () => process(this.responseText);
xhr.open( "GET", "playlist.json");
xhr.send();
```

This example shows how an app: URL can be used in conjunction with a HTTP `Range` header to request a range of bytes from a file inside a package.

Example 4

```
var url = "sample.mp3";
var xhr = new XMLHttpRequest();
xhr.open('GET', url, true);
xhr.responseType = "arraybuffer";
xhr.setRequestHeader('Range', 'bytes=100-199');
xhr.send();
console.log(Uint8Array(xhr.response).byteLength === 100);// true
```

D.10. C. ACKNOWLEDGMENTS

The bulk of the text in this specifications was derived from the *Widget URI scheme* specification. The Systems Application working group acknowledge

the hard work of the Web Applications Working Group in laying down the foundations for this specification.

D.11. D. REFERENCES

D.11.1. D.1 Normative references

[HTTP11]
R. Fielding et al. *Hypertext Transfer Protocol - HTTP/1.1*. June 1999. RFC. URL: http://www.ietf.org/rfc/rfc2616.txt

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt

[SNIFF]
Gordon P. Hemsley. *MIME Sniffing Standard*. Living Standard. URL: http://mimesniff.spec.whatwg.org/

[URL]
Anne van Kesteren. *URL Standard*. Living Standard. URL: http://url.spec.whatwg.org/

D.11.2. D.2 Informative references

[HTML]
Ian Hickson. *HTML*. Living Standard. URL: http://www.whatwg.org/specs/web-apps/current-work/

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

E. APPLICATION LIFECYCLE AND EVENTS

E.1. A SERVICE WORKERS EXTENSION SPECIFICATION

W3C Editor's Draft 16 May 2014

This version:
http://www.w3.org/2012/sysapps/app-lifecycle/

Latest published version:
http://www.w3.org/TR/app-lifecycle/

Latest editor's draft:
http://www.w3.org/2012/sysapps/app-lifecycle/

Editors:
Anssi Kostiainen, Intel
Kenneth Rohde Christiansen, Intel

This specification extends `ServiceWorkerGlobalScope` [[!service-workers]] with APIs for managing the lifecycle of an application and associated events.

E.2. INTRODUCTION

The extensions to the service worker global execution context defined in this specification allow web developers to author applications that manage the application lifecycle and react to system events. These capabilities allow application developers to create applications that integrate closely with the underlying system.

Using the APIs defined in this specification, an application is able to run application logic independently of any user interface scripts and react to:

- Changes in the application lifecycle such as launch and terminate (application events)
- Events sent by the system (system events)
- Scheduled wakeup calls

There is only one class of product that can claim conformance to this specification: a *user agent*.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [[WEBIDL]], as this specification uses that specification and terminology.

E.2.1. Dependencies

This specification relies on the following specifications:

- Service Workers [[!service-workers]] (see also the ServiceWorker GH repo)
- Task Scheduler [[!TASKSCHEDULER]]

E.3. TERMINOLOGY

The term *JavaScript global environment* refers to the global environment concept defined in [[!ECMA-262]].

The *EventHandler* interface represents a callback used for event handlers as defined in [[!HTML5]].

The concepts *queue a task* and *fire a simple event* are defined in [[!HTML5]].

The terms *event handlers* and *event handler event types* are defined in [[!HTML5]].

The `ServiceWorkerGlobalScope` interface is defined in [[!service-workers]]

The `TaskScheduler` interface is defined in [[!TASKSCHEDULER]].

E.4. USE CASES AND REQUIREMENTS

Below is a list of use cases derived from the initial input to the System Applications Working Group Charter that were not addressed by the Runtime and Security Model for Web Applications [[SYSAPPS-RUNTIME]] — a proposal the group decided to obsolete due to lack of implementers' interest. Currently, the domain initially covered by [[SYSAPPS-RUNTIME]] is split across multiple specifications and the use cases compatible with the Web security model are being addressed by a set of specifications: Application Lifecycle and Events and its normative dependencies Service Workers [[!service-workers]], Task Scheduler [[!TASKSCHEDULER]], and the Manifest for web applications [[!appmanifest]]. The set of specifications is expected to grow over time to cover more of the domain and use cases.

In the context of use cases, *main document* refers to a JavaScript global environment.

E.5. A SINGLE ENTRY POINT TO THE APPLICATION

The main document is the main entry point of the application to the system. When loading the main document, the runtime does not display it to the user. If the application intends to show a user interface it has to create windows or interact with the platform in other ways such as by using a notification system.

The runtime can unload the main document in certain circumstances, which results in termination of the application. When the main document is not executing any script, has no pending callbacks, and no open windows, the runtime can decide to unload the main document. In addition, the runtime unloads the main document in order to reduce resource consumption. For example, after loading the main document and no window is visible, the application can be terminated by the runtime.

E.6. BEHAVIOR ADAPTATION AT LAUNCH

This section is non-normative.

The runtime does not create any visible windows by itself when launching an application. This is up to the application and is handled as part of the *launch* event. When launched, the application will know the reason for the launch. The reason could be a scheduled wake up, a persistent event of interest or a direct launch either by the system, another application, or initiated by the user. This allows the application to act differently depending on the reason.

E.6.1. System Event-initiated Launch

The runtime starts the application for the purpose of delivering events from the system. For example, a system-level service might ask an email application to send an email on its behalf. To handle these cases, the runtime will listen to system events in order to launch the main document in response to them.

An application subscribes to system events either at install time or at runtime.

E.6.2. Wakeup-initiated Launch

This section is non-normative.

An application can schedule itself for wake-up by scheduling a task. A task will ensure that the application is running and that the launch event has been fired at the main document before resolving the the task.

Note that in order to run a task at the scheduled time, the application can be started a bit earlier.

E.7. TERMINATION SEQUENCE

This section is non-normative.

The runtime terminates the application if it becomes idle or in case of resource constraints.

Before actual termination, the *terminate* event is sent, giving the application the ability to clear up, store state and close windows. In case that the application does not terminate within a given time, the runtime can consider the application as too slow or hanging and has the ability to terminate it immediately (forced termination). The application can save its state periodically to protect against data loss, in such a case.

After the *terminate event*, or in case of forced termination, the runtime will progress to actual termination, which at least includes closing all remaining windows and unloading of the main document. What other resources the runtime unloads is up to the implementation.

There is an exception as the application can receive a wakeup or a system event while processing the terminate event. If this is the case, the terminate event is followed by a *terminate canceled* event, as actual termination will not happen.

The runtime will avoid terminating an active, focused application if at all possible. However, the runtime can terminate the application as a last resort (e.g., due to resource exhaustion or bad behavior).

The runtime prevents an application from interfering with the application's termination, e.g. event listeners or long-running scripts using APIs such as Geolocation, setTimeout, XMLHttpRequest will not block the runtime from terminating the application.

E.7.1. Application Events

This section is non-normative.

The runtime context associated with the main document can be launched and terminated either by the user, another application, or by the system. The system can decide to terminate an application as long as it is considered idle (no visible views or connected message ports). In the case the application received an event (or something similar, like a task was scheduled) while processing the *terminate* event, the *terminate canceled* event will be fired successively and the main document will stay active until considered idle again.

Mozilla would like the Mozilla Push API to use a [model similar to what is proposed in this spec].

When the application is launched, the launch event is fired with a reason which can be of type *pending event*, *scheduled* or *other*. If the application was launched in order to handle an event it subscribed to or in response to a scheduled task, the type will be *pending event* and *scheduled* respectively. This allows for the application to avoid creating any user interface not resulting from the respective event handling.

Explain how the *other* reason is handled by application developers.

In the case the user or another application started the application, the reason is set to `other`, allowing the main document to load the default user interface and potentially a “screenshot” of the application while the main user interface is being built in the background.

For handling application lifecycle, the application's main document can listen to the events.

E.8. SYSTEM EVENTS

This section is non-normative.

A *system event* is an event sent by the system. This event type does not originate from the DOM itself and thus lives outside of the main document's lifetime. A system event can wake up a terminated application. An implementation can also allow a system event to wake up the system from sleeping.

A typical browser-driven use case for system events is an email application that wants to show a desktop notification when a push notification is received so the user is informed that there are new emails even though the tab in which the email application was running has been closed.

System events are applicable to event types that fire at low frequency and support filtering. For example, they are not appropriate for user interaction events that require a visible user interface or fire frequently.

When an application is launched in response to a system event, its main document is loaded, a launch event is dispatched, and immediately after the launch event, the system event handler is called. If the application did not register the listener as part of the launch event, nothing happens.

To register for system events, need an event handler that is triggered when the application is first installed, the application is updated, or the runtime is updated to a new version ("oninstalled" or similar). Registration through the Manifest would be beneficial too.

E.8.1. Filtered Events

This section is non-normative.

Filtered events are a mechanism that allows listeners to specify a subset of events that they are interested in. A listener that makes use of a filter is not invoked for events that do not pass the filter, which makes the listening code more declarative and efficient.

To prevent an application from being woken up for no reason, filtering happens in the runtime and not in an application.

The difference between "persistent" and "regular" DOM events must be made obvious so that developers do not expect regular DOM events to behave similarly.

E.8.2. Event Registration

This section is non-normative.

When an application subscribes to an event, it will be subscribed to it until it is unsubscribed, the application is uninstalled, updated or terminated.

The listeners only exist in the context of the main document. Event listeners for system events need to be registered each time the main document is launched after termination.

E.9. CREATING WINDOWS

This section is non-normative.

One or more windows might be created from the main document. These windows are directly scriptable by the main document.

TODO: expand windowing use cases in a separate specification if needed.

E.10. REQUIREMENTS

Below is a summary of requirements derived from the above use cases:

1. An application (e.g. a background service) MUST be able to run without visible user interface.
2. An application MUST be able to decide when to show the user interface, if at all. It is up to the application developer to decide when it is appropriate to show the user interface.
    - An application MUST be able to show the user interface only after it is fully constructed with the right dimensions and all the needed data has been loaded.
3. The runtime model MUST support authoring an application (or a service without user interface) that can be terminated without user's consent, and that is able to restore to its previous state.
4. After being launched, an application MUST be able to execute scripts to recreate its state before recreating the actual user interface.
5. An application MUST be able to show a different user interface given how the app was launched.
    - For example, if launched as a photo picker, the application will not show the default application window, but instead creates a special purpose user interface.
6. The runtime MUST provide a mechanism to prevent an application from being launched unnecessarily.
    - As the system events can result in launching dormant apps, it is important that that only happens for subscribed events which support pre-filtering. For example, if an application listens to a "USB plugged" event, it can additionally ask to only listen to a specific device connected or a specific port.
7. The application MUST be able to enumerate windows associated with it, and create new windows.
8. The application MUST be able to create a window and have it laid out correctly with the right dimensions before being shown.
    - This allows emulating the splash screen/application screenshot at launch for any screen size, before loading any application logic, so that the screenshot is not needed to be part of the manifest.

E.11. APIS AVAILABLE TO SERVICE WORKERS

An environment through which the interfaces defined in this specification are exposed to JavaScript is referred to as the *service worker global execution context* (also referred to as the *global execution context of a Service Worker* in [[!service-workers]]) whose global object is referred to as *service worker global scope*.

E.11.1. Extensions to the `ServiceWorkerGlobalScope` interface

attribute EventHandler onlaunch

attribute EventHandler onterminate

attribute EventHandler onterminatecanceled

readonly attribute TaskScheduler taskScheduler

This is a (non-exhaustive) list of features `ServiceWorkerGlobalScope` inherits from `WorkerGlobalScope`:

- `navigator` object
- `location` object (read-only)
- `XMLHttpRequest()` method
    - If the JavaScript global environment is a worker environment, the responseType of document is not supported as per [[XHR]].
- `setTimeout()/clearTimeout()` and `setInterval()/clearInterval()`
- `applicationCache` object
- `importScripts()` method
- `Worker()` method (spawning web workers)
- `indexedDB` object
- ...

Communicating with the service worker is done with explicit `MessagePort` objects similar to shared workers. ServiceWorkerGlobalScope [[!service-workers]] has a `clients` attribute, which represents a list of windows or workers that match the service worker's origin and scope.
Remove features that do not have strong use cases and consider them in v2. After implementation feedback, we can add features that appear to be lacking. For example, `reason` in `LaunchEvent`, `terminate` and `terminatecanceled` are proposed to be deferred to v2 without strong use cases.

When the *application is launched*, the user agent MUST queue a task to launch the application.

When the *application is terminated*, the user agent MUST queue a task to terminate the application.

When the *application termination is canceled*, the user agent MUST queue a task to cancel the termination.

The following are the event handlers (and their corresponding event handler event types) that MUST be supported, as event handler IDL attributes, by all objects implementing the `ServiceWorkerGlobalScope` interface:

event handler	event handler event type
`onlaunch`	`launch`
`onterminate`	`terminate`
`onterminatecanceled`	`terminatecanceled`

It would be nice to have a diagram that shows when all events are fired and the order.

E.11.2. Launching the Application

readonly attribute DOMString reason

attribute DOMString reason

When the user agent is REQUIRED to *launch the application*, the user agent MUST run the following steps:

1. Re-instantiate the pre-existing version of the service worker global execution context, if any. Otherwise, establishing a new service worker global execution context.
2. Create a LaunchEvent object and initialize it with the given name `launch`.
3. Initialize the *`reason`* attribute to a value corresponding to a launch reason as defined in the table below.
4. Dispatch the newly created `LaunchEvent` object at the `ServiceWorkerGlobalScope` object.

`reason` attribute value	Description	Times [fired]	When
`pending-event`	The application is launched in response to a system event it is listening to.	Zero or more.	
`scheduled`	The application is launched by the system at the scheduled time.	Zero or more.	
`other`	Other reason.	Zero or more.	

The pending-event, scheduled or other event types are confusing. This should be implied by the event "class" or the event name.

Need a mechanism for passing data to the service worker from the application that launched it. Does Web Activities/Intents address the use case (in scope for Web Intents Task Force), or is this addressed by e.g. passing the data in `LaunchEvent`?

E.11.3. Terminating the Application

When the user agent is REQUIRED to *terminate the application*, it MUST run the following steps:

1. Fire a simple event named terminate at the `ServiceWorkerGlobalScope` object.
2. Spin the event loop for a user-agent-defined amount of time.
   This is intended to allow the application to run scripts to persist state, do clean up tasks before being terminated.
3. Close all the windows created by the service worker script.
4. Discard the service worker.

Need to make it clear that the service worker can be terminated only when it is idling, all Promises resolved, all indexedDB transactions completed, no Workers running etc. However, Badly behaving application that try to prevent an application for being closed can be killed by the system similarly to `onunload`.

E.11.4. Canceling the Termination

readonly attribute DOMString reason

attribute DOMString reason

When the user agent is REQUIRED to *cancel the termination*, the user agent MUST run the following steps:

1. Cancel the already-running instance of the terminate the application algorithm, if any.
2. Create an event that uses the `TerminateCanceledEvent` interface, with the name `terminatecanceled`.
3. Initialize the `reason` attribute to a value corresponding to a terminate cancellation reason as defined in the table below.
4. Dispatch the newly created `TerminateCanceledEvent` object at the `ServiceWorkerGlobalScope` object.

`reason` attribute value	**Description**	**Times [fired]**	**When**
-	-	-	-
-	-	-	-

E.12. ACKNOWLEDGMENTS

Thanks to everyone who have contributed to the Service Worker proposal that provides primitives for this specification to build atop.

Some use cases are derived from Adam Barth’s execution model proposal referenced in the System Applications Working Group Charter. Thanks to the Chrome team for their experiments with Packaged Apps. Also special thanks to Thiago Marcos P. Santos and Caio Marcelo de Oliveira Filho for their comments.

Also, big thank you to all SysApps Toronto participants who reviewed the proposal, sent feedback and participated in the task force session.

F. TASK SCHEDULER API

W3C Editor Draft — 13 October 2014

This version:
http://www.w3.org/2012/sysapps/web-alarms/

Participate:
public-sysapps@w3.org (archives)

File a bug

Latest published version:
http://www.w3.org/TR/web-alarms/

Latest editor's draft:
http://www.w3.org/2012/sysapps/web-alarms/

Previous versions:
http://www.w3.org/TR/2013/WD-web-alarms-20130205/

Editors:
Mahesh Kulkarni, Samsung Electronics, Co., Ltd, mahesh.kk@samsung.com

Former Editors:
Christophe Dumez, representing Intel and Samsung Electronics (Until January 2013 and mid-August 2014, respectively)

F.14. ABSTRACT

This specification defines an API to schedule a task at a specified time. When the indicated time is reached, the application that scheduled the task will be notified via a functional event on a service worker. A task event will be delivered to a service worker, regardless of whether the application is active on user agent. Applications such as an alarm clock or an auto-updater may utilize this API to perform certain action at a specified time.

F.15. TABLE OF CONTENTS

F.16. 1 INTRODUCTION

This section is non-normative.

Example use of the ScheduledTask API for adding, getting and removing and listening for the alarm clock use cases:

How to set an alarm 10 minutes from now?

```
    // https://example.com/serviceworker.js
    this.ontask = function(task) {
        alert(task.data.message);
        console.log("Task scheduled at: " + new Date(task.time));
        // From here on we can write the data to IndexedDB, send it
        // to any open windows, display a notification, etc.
    }

    // https://example.com/webapp.js
    function onTaskAdded(task) {
        console.log("Task successfully scheduled.");
    }

    function onError(error) {
        alert("Sorry, couldn't set the alarm: " + error);
    }
    navigator.serviceWorker.ready.then(function(serviceWorkerRegistration) {
        serviceWorkerRegistration.taskScheduler.add(Date.now() + (10 * 60000), {
            message: "It's been 10 minutes, your soup is ready!"
        }).then(onTaskAdded, onError);
    });
```

How to get all the scheduled tasks whose time is in the future?

```
    navigator.serviceWorker.getRegistration().then(function(registration) {
        registration.taskScheduler.getPendingTasks().then(function(tasks) {
            alert("There are " + tasks.length + " tasks set.");
        }, function(error) {
            alert("An error occurred getting the scheduled tasks.");
        });
    }, function(error) {
        alert("An error occurred getting the scheduled tasks.");
    });
```

How to remove a scheduled task?

```
    navigator.serviceWorker.getRegistration().then(function(registration) {
        var request = registration.taskScheduler.remove(id).then(function() {
                alert("Task removed");
            }, function(error) {
                alert("Sorry, can't remove the task.");
            });
    }, function(error) {
        alert("An error occurred getting the scheduled tasks.");
    });
```

F.17. 2 CONFORMANCE

This specification defines conformance criteria for a single product: the *user agent* that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the *Web IDL specification* [WEBIDL], as this specification uses that specification and terminology.

F.18. 3 TERMINOLOGY

A *JSON-serializable object* is an `object` that when serialized or stringified conforms to the JSON Grammar as defined in [ECMASCRIPT].

The *`EventHandler`* interface represents a callback used for handling events as defined in [HTML5].

The *`Promise`* interface provides asynchronous access to the result of an operation that is ongoing, has yet to start, or has completed, as defined in [ECMASCRIPT6].

The concepts *queue a task*, *event handler IDL attribute* and *fire a simple event* are defined in [HTML5].

The concepts *event* and *fire an event named e*are defined in [DOM].

The terms *event handler* and *event handler event types* are defined in [HTML5].

Service worker, *service worker registration*, *ServiceWorker*, *ServiceWorkerRegistration*, *ServiceWorkerGlobalScope*, *ExtendableEvent*, and *Handle Functional Event* are defined in [SERVICE-WORKERS].

F.19. 4 REQUIREMENTS

Below is a summary of requirements associated with this API:

1. An application *must* only be able to access its own scheduled tasks.
2. A scheduled task identifier *must* be unique within the application origin.
3. A scheduled task *must* persist if the system is restarted.
4. A scheduled task *must* actively wake the system if the scheduled time is reached while sleeping.
5. A scheduled task that was missed (e.g. because the device was off or the clock jumped past it) should be fired as soon as possible.
6. A scheduled task and its associated data *must* be removed when the application's service worker registration is uninstalled.

F.20. 5 TASK SCHEDULER API

This section is non-normative.

The task scheduler supports the following features:

- Web applications can schedule multiple tasks and get a returned ID for each of them.
- Each `ScheduledTask` has a unique identifier that can be used to specify and remove the scheduled task.
- Web applications can pass a JSON-serializable object to describe more details about each task setting.
- When a scheduled time is reached, an `task` event is sent to the application.
- ScheduledTask API actually does more than `setTimeout()` because it can actively *wake* the system from sleeping and scheduled task are not lost when closing the application or restarting the system.

F.20.1. 5.1 Interface ServiceWorkerRegistration

The Service Worker specification defines a ServiceWorkerRegistration interface [SERVICE-WORKERS], which this specification extends.

```
partial interface ServiceWorkerRegistration {
    readonly attribute TaskScheduler taskScheduler;
}
```

The *taskScheduler* attribute provides the developer access to a TaskScheduler.

F.20.2. 5.2 Interface TaskScheduler

The TaskScheduler interface exposes methods to get, set or remove scheduled tasks. ScheduledTasks are application specific, so there is no way to see the tasks scheduled by other applications nor to modify them. Developers should set an ontask event handler in the associated service worker to listen for the task event when scheduled tasks should be executed.

```
interface TaskScheduler {
    Promise getPendingTasks();
    Promise add(DOMTimeStamp time, optional any data);
    Promise remove(DOMString taskId);
};
```

When invoked, the *getPendingTasks()* method *must* run the following steps:

1. Make a request to the system to retrieve the tasks that were registered by the current application and whose scheduled time is in the future.
2. Let *promise* be a new Promise object and *resolver* its associated resolver.
3. Return *promise* and run the remaining steps asynchronously.
4. If an error occurs, run these substeps and then terminate these steps:
    1. Let *error* be a new DOMException exception whose name is the same as the error returned.
    2. Run *resolver*'s internal *reject* algorithm with *error* as value.
5. When the operation completes successfully, run these substeps:
    1. Let *tasks* be a new array containing the ScheduledTask objects that were retrieved.
    2. Run *resolver*'s intenal fulfill algorithm with *tasks* as value.

When invoked, the *add(time[, data])* method *must* run the following steps:

1. Make a request to the system to schedule a new task for the current application that will trigger at the given time (number of milliseconds since the epoch). If the time argument is in the past, the task will be executed as soon as possible, asynchronously. The system *must* associate the JSON-serializable data with the task if provided.

2. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
3. Return *promise* and run the remaining steps asynchronously.
4. If an error occurs, run these substeps and then terminate these steps:
    1. Let *error* be a new `DOMException` exception whose `name` is `"QuotaExceededError"` if the `data` argument exceeds an implementation-dependent size limit, or whose `name` is the same as the error returned otherwise.
    2. Run *resolver*'s internal *reject* algorithm with *error* as `value`.
5. When the operation completes successfully, run these substeps:
    1. Let *task* be a new `ScheduledTask` object.
    2. Set *task*'s `id` attribute to the unique identifier returned by the system for the newly registered task.
    3. Set *task*'s `time` attribute to the `time` argument.
    4. Set *task*'s `data` attribute to the `data` argument, if provided.
    5. Run *resolver*'s internal `fulfill` algorithm with *task* as `value`.

When invoked, the *`remove(taskId)`* method *must* run the following steps:

1. Make a request to the system to unregister the task with the given unique `taskId` identifier.
2. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
3. Return *promise* and run the remaining steps asynchronously.
4. If an error occurs, run these substeps and then terminate these steps:
    1. Let *error* be a new `DOMException` exception whose `name` is the same as the error returned.
    2. Run *resolver*'s internal *reject* algorithm with *error* as `value`.
5. When the operation completes successfully, run these substeps:
    1. Let *removed* be a boolean value.
    2. Set *removed* to `true` if the task was removed, and to `false` if there was no task with the given identifier.
    3. Run *resolver*'s intenal `fulfill` algorithm with *removed* as `value`.

F.20.3. 5.3 Interface ScheduledTask

The `ScheduledTask` interface captures the properties of a scheduled task.

```
interface ScheduledTask {
    readonly attribute DOMString id;
    readonly attribute DOMTimeStamp time;
    readonly attribute any data;
};
```

The *`id`* attribute returns an identifier for the given `ScheduledTask` object that is unique within the origin. An implementation *must* maintain this identifier when a `ScheduledTask` is added.

The *time* attribute is the time at which this task is scheduled to fire, in milliseconds past the epoch (e.g. Date.now() + n). Due to performance, the task may be delayed past this time.

The *data* attribute optionally represents the JSON-serializable data associated with the task.

F.21. 6 EVENTS

The Service Worker specification defines a ServiceWorkerGlobalScope interface [SERVICE-WORKERS], which this specification extends.

```
partial interface ServiceWorkerGlobalScope {
    attribute EventHandler ontask;
};
```

F.21.1. 6.1 Event Handler

The following is the event handler (and its corresponding event handler event type) that must be supported as attribute by the ServiceWorkerGlobalScope object.

event handler	event handler event type
ontask	task

F.21.2. 6.2 The TaskEvent Interface

The TaskEvent interface represents a scheduled task.

```
interface TaskEvent : ExtendableEvent {
    readonly attribute ScheduledTask task;
};
```

F.21.3. 6.3 Firing task event to service worker

A task event is fired when a scheduled task should be executed. The scheduled task is originated from the system and will wake up a service worker if it is not currently running.

When the scheduled task *task* went off by the system, the user agent must (unless otherwise specified) run these steps:

1. Let *callback* be an algorithm that when invoked with a *global*, fires a service worker task event named task given *task* on *global*.
2. Then run Handle Functional Event with *task*'s service worker registration and *callback*.

To *fire a service worker task event named e* given *task*, fire an event named e with an event using the `TaskEvent` interface whose `task` attribute is initialized to a new `ScheduledEvent` object representing *task*.

F.22. REFERENCES

[B2G-ALARM]
B2G Alarm API Specification, Mounir Lamouri, Kan-Ru Chen and Jonas Sicking. Mozilla.

[DOM]
DOM, Anne van Kesteren, Aryeh Gregor and Ms2ger. WHATWG.

[ECMASCRIPT]
ECMAScript Language Specification. ECMA.

[ECMASCRIPT6]
ECMAScript Language Specification (6th edition, draft). ECMA.

[HTML5]
HTML5, Ian Hickson. W3C.

[SERVICE-WORKERS]
Service Workers, Alex Russell and Jungkee Song. W3C.

[WEBIDL]
Web IDL, Cameron McCormack. W3C.

F.23. ACKNOWLEDGMENTS

We would like to thank Kan-Ru Chen, Mounir Lamouri, Gene Lian and Jonas Sicking for their work on the API design, as well as the WebAPI/B2G teams at Mozilla [B2G-ALARM].

G. CONTACTS MANAGER API

W3C Editor's Draft 04 April 2014

This version:
http://www.w3.org/2012/sysapps/contacts-manager-uri/

Latest published version:
http://www.w3.org/TR/contacts-manager-uri/

Latest editor's draft:
http://www.w3.org/2012/sysapps/contacts-manager-uri/

Editors:
Eduardo Fullea, Telefonica
Jose M. Cantera, Telefonica
Christophe Dumez, Samsung Electronics, Co., Ltd

This specification defines a System Level API which offers a simple interface to manage user's contacts stored in the system's address book. A typical use case of the Contacts API is the implementation of an application to manage said address book.
This document defines a System Level API to manage the user's contacts that are stored in the system's address book. Future versions of this specification are expected to align the contact data model with the Contacts API being defined by the Device APIs Working Group.

If you find any issue with this specification, please file a bug on Github.

G.1. INTRODUCTION

The Contacts API allows to manage (create, edit, remove, etc) user's contacts stored in the system's address book, and thus provides the functionality needed to implement an application to manage said address book.

An example of use is provided below:

```
         var contactName = new ContactName({
           givenNames: ['John'],
           familyNames: ['Doe']
         });
        var mobilePhone = new ContactTelField({ types: ['home'],
                       preferred: true, value: '+34698765432' });
        var contact = new Contact({
           name: contactName,
           phoneNumbers: [mobilePhone]
        });
        navigator.contacts.save(contact).then(
          function(contact) { window.console.log('Contact  ' +
                      contact.name.givenNames[0] + ' ' +
                      contact.name.familyNames[0] + ' saved!'); },
       function(error) {   window.console.error('Error: ' + error); } )
```

This specification defines conformance criteria that apply to a single product: the *user agent* that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [[!WEBIDL]], as this specification uses that specification and terminology.

G.2. TERMINOLOGY

The `EventHandler` interface represents a callback used for event handlers as defined in [[!HTML5]].

The concepts *queue a task* and *fire a simple event* are defined in [[!HTML5]].

The terms *event handler* and *event handler event types* are defined in [[!HTML5]].

The *Promise* interface, the concepts of a *resolver*, a *resolver's fulfill algorithm* and a *resolver's reject algorithm* are defined in [[DOM4]].

G.3. SECURITY AND PRIVACY CONSIDERATIONS

This API must be only exposed to trusted content

G.4. NAVIGATOR INTERFACE

readonly attribute ContactsManager contacts

The object that exposes the contacts management functionality.

G.5. CONTACTSMANAGER INTERFACE

The ContactsManager interface exposes the contacts management functionality.

Promise find ()

This method allows to search contacts within the address book that match the criteria indicated in the `options` parameter. It returns a `Promise` that will allow the caller to be notified about the result of the operation.

optional ContactFindOptions options
Set of criteria that a contact needs to match to be included in the outcomes of the `find` operation.

Promise clear ()
This method allows to remove all contacts in the address book. It returns a `Promise` that will allow the caller to be notified about the result of the operation.

Promise save ()
This method allows to save a contact in the address book, e.g. an existing contact after having been edited. So if a contact with the same identifier, i.e. `id`, already exists in the address book, it will be updated. It returns a `Promise` that will allow the caller to be notified about the result of the operation.
Contact contact
The `Contact` object that is requested to be saved in the address book.

Promise remove ()
This method allows to remove a contact from the address book. It returns a `Promise` that will allow the caller to be notified about the result of the operation.
DOMString contactId
The identifier of the `Contact` object that is requested to be removed from the address book.

attribute EventHandler oncontactschange
May be used to set an event handler to be called when contacts are added, deleted or changed in some way.

The *`find`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
2. Return *promise* and continue the following steps asynchronously.
3. Make a request to the system to retrieve the contacts in the address book matching the criteria indicated in the `options` parameter.
4. If there is an error invoke *resolver*'s reject algorithm with no argument and terminate these steps.
5. When the request has been completed:
    1. Let *contacts* be a new array of `Contact` objects providing the results of the find operation.
    2. Invoke *resolver*'s fulfill algorithm with *contacts* as the `value` argument.

The *`clear`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
2. Return *promise* and continue the following steps asynchronously.

3. Make a request to the system to clear all the contacts in the address book.
4. If there is an error invoke *resolver*'s reject algorithm with no argument and terminate these steps.
5. When the request has been completed invoke *resolver*'s fulfill algorithm with no argument and terminate these steps.

The *save* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
2. Return *promise* and continue the following steps asynchronously.
3. Make a request to the system to save in the address book the `Contact` object passed as parameter.
4. If there is an error invoke *resolver*'s reject algorithm with no argument and terminate these steps.
5. When the request has been completed:
   1. Let *contact* be the `Contact` object as returned by the system, and which therefore has its `id` and `lastUpdated` parameters set, whereas they could be null in the original `Contact` object passed as parameter, for instance if it is a brand new `Contact` object not yet stored in the address book.
   2. Invoke *resolver*'s fulfill algorithm with *contact* as the `value` argument.

The *remove* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
2. Return *promise* and continue the following steps asynchronously.
3. Make a request to the system to remove the `Contact` object identified by the `contactId` passed as parameter from the address book.
4. If there is an error invoke *resolver*'s reject algorithm with no argument and terminate these steps.
5. When the request has been completed invoke *resolver*'s fulfill algorithm with no argument and terminate these steps.

Upon a change in a contact or set thereof is performed (i.e. contact(s) added / modified / removed) the system MUST fire a simple event named `contactschange` that implements the `ContactsChangeEvent` interface at the `ContactsManager` object

G.6. EVENT HANDLERS

The following are the event handlers (and their corresponding event handler event types) that MUST be supported as attributes by the ContactsManager object:

event handler	event handler event type
`oncontactschange`	*`contactschange`*

The event named `contactschange` MUST implement the ContactsChangeEvent interface.

G.7. CONTACTFINDOPTIONS DICTIONARY

The ContactFindOptions dictionary represents the criteria used to select the contacts to be returned by a `find` operation.

DOMString value
Represents the value used for the filtering (e.g. "Tom").

FilterOperator operator
Represents the filtering operator used for the filtering.

sequence<DOMString> fields
Represents the set of fields in which the search is performed (e.g. "givenName").

sequence<DOMString> sortBy
Represents the fields by which the results of the search are sorted (e.g. ["givenNames", "familyNames"]). The results are sorted by the first field, then by the following one, if present, and so on.

SortOrder sortOrder
Represents the order in which the results of the search are sorted.

unsigned long resultsLimit
Represents the maximum number of results that can be returned by the search operation.

G.8. ENUMERATIONS

There are 2 possible filter operations:

contains
With this operation, a contact will be returned if its filtered field contains the given value.

is
With this operation, a contact will be returned if its filtered field matches exactly the given value.

The returned contacts can be sorted in one of the following orders:

ascending
The contacts will be sorted in ascending order.

descending
The contacts will be sorted in descending order.

It is for further study whether the level of flexibility of the filters needs to be increased and/or additional mechanisms need to be put in place so that applications can keep a local copy of the address book and perform the filtering locally (e.g. startTrackingChanges() and getNewChanges() methods)

G.9. CONTACTFIELD INTERFACE

The ContactField interface represents a user's attribute and the types associated to it.

attribute DOMString[] types
Indicates the types of this contact field (e.g. "home", "work").

attribute boolean preferred
Indicates whether this is the preferred contact field.

attribute DOMString value
A string that contains the user's address.

G.9.1. ContactFieldInit Dictionary

sequence<DOMString> types

boolean preferred

DOMString value

G.10. CONTACTTELFIELD INTERFACE

The ContactTelField interface represents a telephone number as well as metadata associated to it, namely the types (e.g. "voice", "text") and the carrier providing service to the telephony subscription associated to that number.

attribute DOMString? carrier
Indicates the carrier providing service to the telephony subscription associated to that number

G.10.1. ContactTelFieldInit Dictionary

DOMString carrier

G.11. CONTACTADDRESS INTERFACE

The ContactAddress interface represents a user's physical address and the types associated to it. This interface is based on vCard 4.0 ADR attribute.

attribute DOMString[] types
Indicates the types of this contact field (e.g. "home", "work").

attribute boolean preferred
Indicates whether this is the preferred contact field.

attribute DOMString streetAddress
A string that contains the name of the street. It maps to the third component in vCard's ADR attribute.

attribute DOMString locality
A string that contains the name of the locality. It maps to the forth component in vCard's ADR attribute.

attribute DOMString region
A string that contains the name of the region. It maps to the fifth component in vCard's ADR attribute.

attribute DOMString postalCode
A string that contains the postal code. It maps to the sixth component in vCard's ADR attribute.

attribute DOMString countryName
A string that contains the name of the country. It maps to the seventh component in vCard's ADR attribute.

G.11.1. ContactAddressInit Dictionary

sequence<DOMString> types

boolean preferred

DOMString streetAddress

DOMString locality

DOMString region

DOMString postalCode

DOMString countryName

G.12. THE CONTACTGENDER ENUM

male
contact is a male.

female
contact is a female.

other
contact has another gender.

none
contact does not have a gender (not applicable).

unknown
contact's gender is unknown.

G.13. CONTACTNAME INTERFACE

The ContactName interface represents a user's naming attributes.

attribute DOMString displayName
A string representing the contact's display name. It maps to vCard's FN attribute.

attribute DOMString[] honorificPrefixes
A string or set thereof representing the contact's honorific prefix(es). It maps to the forth component in vCard's N attribute.

attribute DOMString[] givenNames
A string or set thereof representing the contact's given name(s). It maps to the second component in vCard's N attribute.

attribute DOMString[] additionalNames
A string or set thereof representing any additional name of the contact. It maps to the third component in vCard's N attribute.

attribute DOMString[] familyNames
A string or set thereof representing the contact's family name(s). It maps to the first component in vCard's N attribute.

attribute DOMString[] honorificSuffixes
A string or set thereof representing the contact's honorific suffix(es). It maps to the fifth component in vCard's N attribute.

attribute DOMString[] nicknames
A string or set thereof representing the contact's nick name(s). It maps to vCard's NICKNAME attribute.

G.13.1. ContactNameInit Dictionary

DOMString displayName

sequence<DOMString> honorificPrefixes

sequence<DOMString> givenNames

sequence<DOMString> additionalNames

sequence<DOMString> familyNames

sequence<DOMString> honorificSuffixes

sequence<DOMString> nicknames

G.14. CONTACT INTERFACE

The Contact interface represents a contact stored in the address book. As a principle the attributes are based on vCard 4.0 and reuse the literal used in that standard. Any naming deviation is mentioned in the description of the corresponding attribute. Whereas this correspondence facilitates the import/export from/to vCard format it should be noted that a vCard import/export API is out of scope of this specification as parsing and serializing can be efficiently done in JavaScript, and libraries are readily available.

readonly attribute DOMString? id
Represents a unique identifier of the contact in the address book.

readonly attribute Date? lastUpdated
A `Date` element representing the date when the contact was last updated.

attribute ContactName name
An object representing the different naming attributes of the contact.

attribute ContactField[] emails
A `ContactField` element or set thereof containing the contact's email address(es). It maps to vCard's EMAIL attribute.

attribute DOMString[] photos
A string or set thereof representing the URI of the photo(s) of the contact. It maps to vCard's PHOTO attribute.

attribute ContactField[] urls
A `ContactField` element or set thereof containing the user's urls (e.g. personal blog). It maps to vCard's URL attribute.

attribute DOMString[] categories
A string or set thereof representing the category or categories associated to the contact (e.g. "family"). It maps to vCard's CATEGORIES attribute.

attribute ContactAddress[] addresses
A `ContactAddress` element or set thereof containing the user's physical address(es). It maps to vCard's ADR attribute

attribute ContactTelField[] phoneNumbers
A `ContactTelField` element or set thereof containing the user's telephone numbers. It maps to vCard's TEL attribute

attribute DOMString[] organizations
A string or set thereof representing the organization(s) the contact belongs to. It maps to vCard's ORG attribute

attribute DOMString[] jobTitles
A string or set thereof representing the contact's job title(s). It maps to vCard's TITLE attribute

attribute Date? birthday
A `Date` element representing the contact's birth date. It maps to vCard's BDAY attribute

attribute DOMString[] notes
A string or set thereof specifying supplemental information or a comment that is associated with the contact. It maps to vCard's NOTE attribute

attribute ContactField[] impp
A `ContactField` element or set thereof containing the user's instant messaging address(es). It maps to vCard's IMPP attribute

attribute Date? anniversary
A `Date` element representing the contact's anniversary. It maps to vCard's ANNIVERSARY attribute

attribute ContactGender gender
A string representing the contact's gender. It maps to the first component of vCard's GENDER attribute.

G.14.1. Steps

The Contact interface's contructor when invoked MUST run the following steps:

1. Let *contact* be a new `Contact` object.
2. Make a request to the system to generate a new unique contact identifier.
3. Set the `id` attribute of *contact* to the generated contact identifier.
4. Set the `lastUpdated` attribute of *contact* to the Date at which the constructor was invoked.
5. Set the other attributes of *contact* to the value of the corresponding element in the `contactInitDict` dictionary, if present.

G.14.2. ContactInit Dictionary

ContactName name

sequence<ContactField> emails

sequence<DOMString> photos

sequence<ContactField> urls

sequence<DOMString> categories

sequence<ContactAddress> addresses

sequence<ContactTelField> phoneNumbers

sequence<DOMString> organizations

sequence<DOMString> jobTitles

Date birthday

sequence<DOMString> notes

sequence<ContactField> impp

Date anniversary

ContactGender gender

G.15. CONTACTSCHANGEEVENT INTERFACE

The ContactsChangeEvent interface represents events related to the set of contacts that have been simultaneously added, removed and/or modified. Changes that are applied to the address book in batches cause a ContactsChangeEvent with multiple contact references to be fired, whereas changes applied sequentially cause a ContactsChangeEvent with a single contact reference to be fired

readonly attribute DOMString[] added
Indicates the identifier(s) of the `Contact` object(s) that have been added.

readonly attribute DOMString[] modified
Indicates the identifier(s) of the `Contact` object(s) that have been modified.

readonly attribute DOMString[] removed
Indicates the identifier(s) of the `Contact` object(s) that have been removed.

G.15.1. ContactsChangeEventInit Dictionary

sequence<DOMString> added

sequence<DOMString> modified

sequence<DOMString> removed

G.16. ACKNOWLEDGEMENTS

The editors would like to express their gratitude to the Mozilla B2G Team for their technical guidance, implementation work and support, and specially to Tantek Çelik and Gregor Wagner, the original authors of B2G Contact API.

H. MESSAGING API

W3C Editor's Draft 29 November 2014

This version:
http://www.w3.org/2012/sysapps/messaging/

Latest published version:
http://www.w3.org/TR/messaging/

Latest editor's draft:
http://www.w3.org/2012/sysapps/messaging/

Editors:
Eduardo Fullea, Telefonica
Jose M. Cantera, Telefonica
Zoltan Kis, Intel

This specification defines a System Level API which offers a simple interface to get access to mobile messaging services. A typical use case of the Messaging API is the implementation of a messaging client application that allows the user to send SMS and MMS messages as well as to access and manage the received SMS and MMS messages.

H.1. INTRODUCTION

The Messaging API provides operations to get access to the primitives offered by mobile messaging services (send, receive) as well as those that allow to manage a mobile messaging client inbox (delete, store, mark as read)

An example of use is provided below:

```
navigator.messaging.sms.send ( '+1234567890', 'How are you?').then(
  function(message) { window.console.log('Message with identifier  ' +
                      message.messageID + ' sent at ' + message.timestamp); },
  function(error) {   window.console.error('Error: ' + error); } )
```

This specification defines conformance criteria that apply to a single product: the *user agent* that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [[!WEBIDL]], as this specification uses that specification and terminology.

H.2. TERMINOLOGY

The `EventHandler` interface represents a callback used for event handlers as defined in [[!HTML5]].

The concepts *queue a task* and *fire an event* are defined in [[!HTML5]].

The terms *event handler* and *event handler event types* are defined in [[!HTML5]].

The *Promise* interface, the concepts of a *resolver*, a *resolver's fulfill algorithm* and a *resolver's reject algorithm* are defined in [[DOM4]].

H.3. SECURITY AND PRIVACY CONSIDERATIONS

This API must be only exposed to trusted content

H.4. NAVIGATOR INTERFACE

readonly attribute Messaging messaging
The object that exposes the interface to mobile messaging services.

H.5. MESSAGINGMANAGER INTERFACE

The MessagingManager interface represents the initial entry point for getting access to the mobile messaging services, i.e. SMS and MMS.

readonly attribute SmsManager sms
Provides access to the SMS service's specific functionality.

readonly attribute MmsManager mms
Provides access to the MMS service's specific functionality.

Promise findMessages ()
This method makes a request to retrieve the messages matching the filter described by the `filter` parameter and according to the filtering options described in the `options`. It returns a new `Promise` that will be used to notify the caller about the result of the operation, which is a MessagingCursor to access the set of messages.

MessagingFilter filter
Filter that identifies the set of messages that are requested to be retrieved

FilterOptions options
Indicates the filtering options (i.e. sorting criteria, sorting order, limit of results).

Promise findConversations ()
This method makes a request to retrieve the list of conversations in which the messages can be grouped using the criteria defined by the `groupBy` parameter. Only those messages matching the filter described in the `filter` parameter SHALL be included in the resulting conversations, what can be useful for instance to filter just a specific type of messages (e.g. SMS) or to implement message search in a conversational messaging client. It returns a new `Promise` that will be used to notify the caller about the result of the operation, which is a MessagingCursor to access the set of conversations.

DOMString groupBy
Indicates the criteria used to define the conversations. It may have the values 'participants' if a conversation is to be defined as the set of messages exchanged among the same set of parties, and 'subject' if a conversation is to be defined as the set of messages with the same subject.

MessagingFilter filter
Filter that identifies the set of messages that are requested to be included in the resulting conversations.

FilterOptions options
Indicates the filtering options (i.e. sorting criteria, sorting order, limit of results) to be aplied when filtering the messages to be included in each of the resulting conversations.

Promise getMessage ()
This method makes a request to retrieve the message identified by the `messageID` parameter. It returns a new `Promise` object which allows the caller to be notified about the result of the operation.

DOMString messageID
Identifier of the message that is requested to be retrieved

Promise deleteMessage ()
This method requests the deletion of the message with identifier equal to the `messageID` parameter. A new `Promise` is returned in order to notify the request result (success or error) to the caller.

DOMString messageID
Identifier of the message that is requested to be deleted

Promise deleteConversation ()

This method requests the deletion of all the messages in the conversation with identifier equal to the `conversationID` parameter. A new `Promise` is returned in order to notify the request result (success or error) to the caller.

DOMString conversationID

Identifier of the conversation whose messages are requested to be deleted

Promise markMessageRead ()

This method requests to mark as read or unread the message with identifier equal to the `messageID` parameter. The method returns a new `Promise` that will allow the caller to be notified about the result (success, error) of the operation.

DOMString messageID

Identifier of the message that is requested to be marked as read or unread

boolean value

Indicates whether the message is to be marked as read ('true') or unread ('false')

optional boolean sendReadReport = false

Indicates that, in case a Read Report was requested, it is to be sent ('true') or not ('false', which is the default)

Promise markConversationRead ()

This method requests to mark as read or unread all the messages in the conversation with identifier equal to the `conversationID` parameter. The method returns a new `Promise` that will allow the caller to be notified about the result (success, error) of the operation.

DOMString conversationID

Identifier of the conversation whose messages are requested to be marked as read or unread

boolean value

Indicates whether the messages in the conversation are to be marked as read ('true') or unread ('false')

optional boolean sendReadReport = false

Indicates that, in case a Read Report was requested for any MMS message in the conversation, a Read Report is to be sent ('true') or not ('false', which is the default)

H.5.1. Steps

The *`findMessages`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.

3. Make a request to the system to get the message(s) matching the filter included in the `filter` parameter and according to the filtering options described in the `options` parameter.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *messagingCursor* be a new `MessagingCursor` object providing access to the results of the retrieval, i.e. the set of `SmsMessage` and/or `MmsMessage` elements.
    2. Invoke *resolver*'s fulfill algorithm with *messagingCursor* as the `value` argument.

The *`findConversations`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Make a request to the system to get the set of conversations in which the messages can be grouped, according to the set of participants or the subject as indicated in the `groupBy` parameter, and filtering the messages included in those conversations according to the filter included in the `filter` parameter and the filtering options described in the `options` parameter.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *messagingCursor* be a new `MessagingCursor` object providing access to the results of the retrieval, i.e. the set of `Conversation` elements.
    2. Invoke *resolver*'s fulfill algorithm with *messagingCursor* as the `value` argument.

The *`getMessage`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Make a request to the system to get the message with identifier equal to the `messageID` parameter passed in the request.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *message* be the `SmsMessage` or `MmsMessage` whose identifier matches the `messageID` parameter.
    2. Invoke *resolver*'s fulfill algorithm with *message* as the `value` argument.

The *`deleteMessage`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Make a request to the system to delete the message with identifier equal to the `messageID` parameter passed in the request.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *messageID* be the `messageID` parameter passed in the request
    2. Invoke *resolver*'s fulfill algorithm with *messageID* as the `value` argument.

The *`deleteConversation`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Make a request to the system to delete the messages in the conversation with identifier equal to the `conversationID` parameter passed in the request.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *conversationID* be the `conversationID` parameter passed in the request
    2. Invoke *resolver*'s fulfill algorithm with *conversationID* as the `value` argument.

The *`markMessageRead`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Make a request to the system to mark as read/unread (depending on the `value` parameter being respectively 'true' or 'false') the message with identifier equal to the `messageID` parameter passed in the request, and to send a Read Report if `sendReadReport` is set to 'true'.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *messageID* be the `messageID` parameter passed in the request
    2. Invoke *resolver*'s fulfill algorithm with *messageID* as the `value` argument.

The *`markConversationRead`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`

2. Return *promise* to the caller.
3. Make a request to the system to mark as read/unread (depending on the `value` parameter being respectively 'true' or 'false') the messages in the conversation with identifier equal to the `conversationID` parameter passed in the request, and in case `sendReadReport` is set to 'true', to send a Read Report for each of the MMS messages for which it was requested.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *conversationID* be the `conversationID` parameter passed in the request
    2. Invoke *resolver*'s fulfill algorithm with *conversationID* as the `value` argument.

It is FFS whether the methods deleteMessage() and markMessageRead() should also accept an array of message identifiers as input parameter.

H.6. SMSMANAGER INTERFACE

The SmsManager interface represents the SMS messaging service manager.

readonly attribute MessageType type

MUST return the type of the messaging service manager. It can have the following values: 'sms' or 'mms'.

readonly attribute DOMString[] serviceIDs

MUST return the identifier of the different services for this type of messaging service (e.g. 'sms_sim1').

Promise segmentInfo ()

This method issues a request to get information on the number of concatenated SMS segments needed to send the text in the `text` parameter, the number of characters available per segment and the maximum number of available characters in the last segment. A `Promise` object will be returned in order to notify the result of the request.

DOMString text

Text intended to be sent as an SMS, whose segmentation information is checked by this method.

optional DOMString serviceID

Identifier of the service through which the message is would be sent.

Promise send ()

This method issues a request to the messaging system to send an SMS message with the text of the `text` parameter to the destination number indicated in the `to` parameter. A `Promise` object will be returned in order to notify the result of the request.

DOMString to

Destination number for the SMS message.

DOMString text

Content of the SMS message to be sent.

optional DOMString serviceID

Identifier of the service through which the message is requested to be sent.

Promise clear ()

This method makes a request to delete all the messages associated to the messaging service passed as parameter.

DOMString serviceID

Identifies the messaging service all whose messages are requested to be deleted.

attribute EventHandler onreceived

Handles the `received` event of type MessagingEvent, fired when a new message is received on this messaging service manager.

attribute EventHandler onsent

Handles the `sent` event of type MessagingEvent, fired when a new message is sent using this messaging service manager.

attribute EventHandler ondeliverysuccess

Handles the `deliverysuccess` event of type DeliveryReportEvent, fired when a new succesful delivery report is received on this messaging service manager.

attribute EventHandler ondeliveryerror

Handles the `deliveryerror` event of type DeliveryReportEvent, fired when a new failure delivery report is received on this messaging service manager.

attribute EventHandler onserviceadded

Handles the `serviceadded` event of type ServiceChangeEvent, fired whenever a new messaging service is enabled on this messaging service manager.

attribute EventHandler onserviceremoved

Handles the `serviceremoved` event of type ServiceChangeEvent, fired when an existing messaging service is disabled on this messaging service manager.

H.6.1. Steps

The `send` method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Let *smsMessage* be a new instance of `SmsMessage`:

   1. Generate a identifier for this message that is globally unique within the implementation, i.e. there cannot be any other message with the same identifier.
   2. Set the `messageID` of *smsMessage* to the generated identifier.
   3. Set the `type` of *smsMessage* to 'sms'.
   4. Set the `serviceID` of *smsMessage* to the identifier of the service used to send the message, i.e. the one passed in the `serviceID` parameter, if provided, or otherwise to the first item in the `serviceIDs` attribute of the `SmsManager`.
   5. Set the `from` of *smsMessage* to the number of the mobile subscription used to send this SMS message.
   6. Set the `read` of *smsMessage* to 'true'.
   7. Set the `to` of *smsMessage* to the value of the `to` parameter.
   8. Set the `body` of *smsMessage* to the value of the `text` parameter.
   9. Set the `messageClass` of *smsMessage* to 'class1'.
   10. Set the `state` of *smsMessage* to 'sending'.
   11. Set the `deliveryStatus` of *smsMessage* to 'pending' if a delivery report has been requested or to 'not-applicable' otherwise.
   12. Make a request to the system to send an SMS message with text passed in the `text` parameter to the number of the recipient indicated in the `to` parameter, using the proper service as described above.
   13. Queue a task to monitor SMS submission process.

4. If an *error* occurs run these substeps and terminate these steps
   1. If a delivery report had been requested set the `deliveryStatus` of *smsMessage* to 'error'.
   2. invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
   1. Set the `state` of *smsMessage* to 'sent'.
   2. Set the `timestamp` of *smsMessage* to the device's date when the SMS message was sent, i.e. when the SMS-SUBMIT Protocol Data Unit was sent.
   3. Invoke *resolver*'s fulfill algorithm with *smsMessage* as the `value` argument.
   4. Queue a task to fire an event named `sent` with the `message` attribute set to *smsMessage*.

The *`segmentInfo`* method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Let *smsSegmentInfo* be a new instance of `SmsSegmentInfo`.
4. Make a request to the system to calculate the segmentation information related to the sending as SMS the text passed in the `text` parameter, using the service with identifier equal to the one passed in the `serviceID` parameter, if provided, or otherwise to the first item in the `serviceIDs` attribute of the `SmsManager`.

5. Queue a task to monitor the calculation process.
6. If an *error* occurs run these substeps and terminate these steps
    1. invoke *resolver*'s reject algorithm with *error* as the `value` argument.
7. When the request has been successfully completed:
    1. Set the `segments` of *smsSegmentInfo* to the number of concatenated SMS segments needed to send the provided text.
    2. Set the `charsPerSegment` of *smsSegmentInfo* to the number of characters available per SMS segment. This number depends on the encoding to be used to send the SMS message, which in turn depends on the language / special characters included in the text.
    3. Set the `charsAvailableInLastSegment` of *smsSegmentInfo* to the maximum number of available characters in the last segment that would be needed to send the input string. This provides useful information to the user on the number of characters that can type without requiring an additional SMS segment to send the text.
    4. Invoke *resolver*'s fulfill algorithm with *smsSegmentInfo* as the `value` argument.

Note that the application that has invoked the `segmentInfo` method SHOULD NOT split the text in a set of strings that fit each into a single SMS segment and send each of them by an independent call to the `sendSMS` method as it would result in different independent SMS messages being sent, but SHOULD instead send the full message in a single `sendSMS` request. However having information on the number of SMS segments may be required by the application in order to inform the user (e.g. in case the length of the text impacts on the price charged for sending the message).

Upon a new SMS message being received, the user agent MUST:

1. Let *smsMessage* be a new instance of `SmsMessage`.
2. Generate a identifier for this message that is globally unique within the implementation, i.e. there cannot be any other message with the same identifier.
3. Set the `messageID` of *smsMessage* to the generated identifier.
4. Set the `type` of *smsMessage* to 'sms'.
5. Set the `serviceID` of *smsMessage* to the identifier of the service at which the message has been received.
6. Set the `from` of *smsMessage* to the sender of the SMS message, i.e. the value of the TP Originating Address (TP-OA) field of the SMS message [[!GSM-SMS]].
7. Set the `timestamp` of *smsMessage* to the timestamp of the SMS message, i.e. the value of the TP-Service-Centre-Time-Stamp (TP-SCTS) parameter received in the SMS DELIVER Protocol Data Unit [[!GSM-SMS]].
8. Set the `read` of *smsMessage* to 'false'.
9. Set the `to` of *smsMessage* to the recipient of the SMS message, i.e. the value of the TP Destination Address (TP-DA) field of the SMS message [[!GSM-SMS]].

10. Set the `messageClass` of *smsMessage* to the message class indicated in the TP-Data-Coding-Scheme (TP-DCS) field of the SMS message [[!GSM-SMS]].
11. Set the `body` element to the text of the received SMS message, i.e. the value of the SM element contained within the TP User Data (TP-UD) field of the SMS message [[!GSM-SMS]].
12. Set the `state` of *smsMessage* to 'received'.
13. Set the `deliveryStatus` of *smsMessage* to 'not-applicable'.
14. Queue a task to fire an event named `received` with the `message` attribute set to *smsMessage*.
15. Queue a task to fire a system message named `received` of type `ReceivedMessage` with the `message` attribute set to *smsMessage*.

Upon a delivery report of a previously sent SMS message being received, the user agent MUST

1. Let *smsMessage* be the instance of `SmsMessage` to which this delivery report is related.
2. Set the `deliveryStatus` parameter of *smsMessage* to 'success' or 'error' depending on the reported result.
3. Set the `deliveryTimestamp` of *smsMessage* to the delivery time of the SMS message, i.e. the TP-Discharge-Time (TP DT) parameter included in the SMS-STATUS-REPORT Protocol Data Unit [[!GSM-SMS]].
4. Queue a task to fire an event named deliverysuccess or deliveryerror respectively if the delivery was successfull or not, with
    1. the `messageID` attribute set to the `messageID` attribute of *smsMessage*,
    2. the `serviceID` attribute set to the `serviceID` attribute of *smsMessage*,
    3. the first item in the `recipients` attribute set to the `to` attribute of *smsMessage*, and
    4. the first item in the `deliveryTimestamps` attribute set to delivery time of such message.
5. Queue a task to fire a system message of type `DeliveryReport`named `deliverysuccess` or deliveryerror respectively if the delivery was successfull or not, with
    1. the `messageID` attribute set to the `messageID` attribute of *smsMessage*,
    2. the `serviceID` attribute set to the `serviceID` attribute of *smsMessage*, and
    3. the first item in the `recipients` attribute set to the `to` attribute of *smsMessage*.
    4. the first item in the `deliveryTimestamps` attribute set to delivery time of such message.

The `clear` method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`

2. Return *promise* to the caller.
3. Make a request to the system to delete all the messages associated to the messaging service with identifier equal to the `serviceID` parameter.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *serviceID* be the `serviceID` parameter passed in the request
    2. Invoke *resolver*'s fulfill algorithm with *serviceID* as the `value` argument.

H.7. EVENT HANDLERS

The following are the event handlers (and their corresponding event types) that MUST be supported as attributes by the SmsManager object.

event handler	event name	event type	short description
`onreceived`	*`received`*	`MessagingEvent`	handles received messages
`onsent`	*`sent`*	`MessagingEvent`	handles sent messages
`ondeliverysuccess`	*`deliverysuccess`*	`DeliveryReportEvent`	handles successful delivery reports
`ondeliveryerror`	*`deliveryerror`*	`DeliveryReportEvent`	handles failure delivery reports
`onserviceadded`	*`serviceadded`*	`ServiceChangeEvent`	handle new messaging services
`onserviceremoved`	*`serviceremoved`*	`ServiceChangeEvent`	handle disabled messaging services

H.8. SMSSEGMENTINFO DICTIONARY

The SmsSegmentInfo dictionary contains information about the segmentation of a given text to be sent as SMS.

long segments

MUST return the total number of SMS segments needed to send the input string, taking into account the encoding to be used to send such message as well as the overhead associated to concatenated SMS messages.

long charsPerSegment

MUST return the number of characters available per SMS segment as per the encoding to be used to send the SMS message. In case the variable length encoding, the value of this element MUST be calculated asumming the minimum length for all the characters.

long charsAvailableInLastSegment

MUST return the maximum number of available characters in the last segment needed to send the input string. In case the variable length encoding, the value of this element MUST be calculated asumming the minimum length for all the remaining characters.

H.9. MMSMANAGER INTERFACE

The MmsManager interface represents the MMS messaging service manager.

readonly attribute MessageType type

MUST return the type of the messaging service manager. It can have the following values: 'sms' or 'mms'.

readonly attribute DOMString[] serviceIDs

MUST return the identifier of the different services for this type of messaging service (e.g. 'sms_sim1').

FetchMode getFetchMode ()

This method requests to retrieve the fetch mode associated to a specific service (the one identified by the `serviceID` parameter, if provided, or the first item in the `serviceIDs` attribute of the `MmsManager` otherwise).

optional DOMString serviceID

Identifier of the service whose fetch mode is queried.

void setFetchMode ()

This method issues a request to the messaging system to set the MMS message fetch mode for the service identified by the `serviceID` parameter, if provided, or for all services otherwise, to the mode indicated in the `fetchMode` parameter.

FetchMode fetchMode

Fetch mode that is requested to be set for a specific service.

optional DOMString serviceID

Identifier of the service whose fetch mode is requested to be set.

Promise send ()

This method issues a request to the messaging system to send an MMS message with the content and recipients included in the `mmsContent` parameter. A `Promise` object will be returned in order to notify the result of the request.

MmsContent mmsContent

Content and recipients of the MMS message to be sent.

optional MmsSendParameters sendParameters

Set of parameters related to the submission of the message (e.g. request of delivery/read report or not).

Promise fetch ()

This method requests to fetch an MMS message with identifier equal to the indicated in the `messageID` parameter from the URL indicated in the MMS notification. The method returns a new `Promise` that will allow the caller to be notified about the result (success, error) of the operation.

DOMString messageID

Identifier of the MMS message that is requested to be download.

Promise clear ()

This method makes a request to delete all the messages associated to the messaging service passed as parameter.

DOMString serviceID

Identifies the messaging service all whose messages are requested to be deleted.

attribute EventHandler onreceived

Handles the `received` event of type MessagingEvent, fired when a new message is received on this messaging service manager.

attribute EventHandler onsent

Handles the `sent` event of type MessagingEvent, fired when a new message is sent using this messaging service manager.

attribute EventHandler ondeliverysuccess

Handles the `deliverysuccess` event of type DeliveryReportEvent, fired when a new succesful delivery report is received on this messaging service manager.

attribute EventHandler ondeliveryerror

Handles the `deliveryerror` event of type DeliveryReportEvent, fired when a new failure delivery report is received on this messaging service manager.

attribute EventHandler onreadsuccess

Handles the `readsuccess` event of type ReadReportEvent, fired when a new succesful read report is received on this messaging service manager.

attribute EventHandler onreaderror

Handles the `readerror` event of type ReadReportEvent, fired when a new failure read report is received on this messaging service manager.

attribute EventHandler onserviceadded
Handles the `serviceadded` event of type ServiceChangeEvent, fired whenever a new messaging service is enabled on this messaging service manager.

attribute EventHandler onserviceremoved
Handles the `serviceremoved` event of type ServiceChangeEvent, fired when an existing messaging service is disabled on this messaging service manager.

It is FFS whether MMS settings (e.g. fetch mode, creation mode) needs to be managed through the MmsManager interface.

H.9.1. Steps

The `send` method when invoked MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. Let *mmsMessage* be a new instance of `MmsMessage` and:
    1. Generate a identifier for this message that is globally unique within the implementation, i.e. there cannot be any other message with the same identifier.
    2. Set the `messageID` of *mmsMessage* to the generated identifier.
    3. Set the `type` of *mmsMessage* to 'mms'.
    4. Set the `serviceID` of *mmsMessage* to the identifier of the service used to send the message, i.e. the one passed in the `serviceID` parameter in `MmsSendParameters`, if provided, or otherwise to the first item in the `serviceIDs` attribute of the `MmsManager`.
    5. Set the `from` of *mmsMessage* to the number of the mobile subscription used to send this MMS message.
    6. Set the `read` of *mmsMessage* to 'true'.
    7. Set the `to` of *mmsMessage* to the `to` in the `mmsContent` parameter.
    8. Set the `cc` of *mmsMessage* to the `cc` array in the `mmsContent` parameter.
    9. Set the `bcc` of *mmsMessage* to the `bcc` array in the `mmsContent` parameter.
    10. Set the `subject` of *mmsMessage* to the value of the the `subject` parameter in `mmsContent`.
    11. Set the `smil` of *mmsMessage* to the value of the the `smil` parameter in `mmsContent`.
    12. Set the `attachments` of *mmsMessage* to the value of the the `attachments` array in `mmsContent` parameter.
    13. Set the `state` of *mmsMessage* to 'sending'.
    14. Add a new item in the `deliveryInfo` attribute of *mmsMessage* for each unique recipient included in the `to`, `cc` and `bcc` parameters (i.e. a single item if the same address has multiple ocurrences across these parameters), with the `recipient` attribute set to

    this recipient's address, with the `deliveryStatus` attribute set to 'pending', if a delivery report has been requested, or 'not-applicable' otherwise and with the `readStatus` attribute set to 'pending', if a read report has been requested, or 'not-applicable' otherwise.
  15. Make a request to the system to send an MMS message with the content passed in the `content` parameter to the number(s) of indicated in the `to` parameter, using the proper service as described above and asking for delivery and/or read report if requested.
  16. Queue a task to monitor MMS sending progress.

4. If an *error* occurs run these substeps and terminate these steps
   1. If a delivery report had been requested set the `deliveryStatus` attribute of the different items in the `deliveryInfo` array attribute of *mmsMessage* to 'error'.
   2. invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
   1. Set the `state` of *mmsMessage* to 'sent'.
   2. Set the `timestamp` of *mmsMessage* to the device's date when the MMS message was sent, i.e. the date when the M-Send.req Protocol Data Unit was sent by the MMS Client.
   3. Invoke *resolver*'s fulfill algorithm with *mmsMessage* as the `value` argument.
   4. Queue a task to fire an event named `sent` with the `message` attribute set to *mmsMessage*.

The reception of an MMS message is a two-step process: Firstly, an MMS notification encapsulated in a WAP Push OTA message [[OMA-PUSH]] is received by the device. This MMS notification contains limited information about the MMS message and a URL where the device can retrieve the full MMS message from. Secondly, the MMS message is retrieved either automatically, i.e. performed right after the reception of the MMS notification, or manually, i.e. invoked manually by the user.

Upon a new MMS notification being received, the user agent MUST:

1. Let *mmsMessage* be a new instance of `MmsMessage`.
2. Generate a identifier for this message that is globally unique within the implementation, i.e. there cannot be any other message with the same identifier.
3. Set the `messageID` of *mmsMessage* to the generated identifier.
4. Set the `type` of *mmsMessage* to 'mms'.
5. Set the `serviceID` of *mmsMessage* to the identifier of the service at which the message has been received.
6. Set the `read` of *mmsMessage* to 'false'.
7. if the fetch mode is manual, never or the device is not in the home network and the the fetch mode is automatic-home:

    1. Set the `from` of *mmsMessage* to the value of the 'From' field of the MMS notification, if present.
    2. Set the `timestamp` of *mmsMessage* to the timestamp of the binary SMS message used to transport the MMS notification, i.e. the value of the TP-Service-Centre-Time-Stamp (TP-SCTS) parameter received in the SMS-DELIVER Protocol Data Unit [[!GSM-SMS]].
    3. Set the `expiry` of *mmsMessage* to the value of the 'X-Mms-Expiry' field of the MMS notification.
    4. Add a new item in the `to` array of *mmsMessage* for each of recipients in the 'To' field of the MMS notification [[!MMS13]], if present.
    5. Add a new item in the `cc` array of *mmsMessage* for each of recipients in the 'Cc' field of the MMS notification [[!MMS13]], if present.
    6. Add a new item in the `bcc` array of *mmsMessage* for each of recipients in the 'Bcc' field of the MMS notification [[!MMS13]], if present.
    7. Set the `subject` attribute to the value of the 'Subject' field of the MMS notification [[!MMS13]], if present.
    8. Set the `state` of *mmsMessage* to 'not-downloaded'.
8. if the fetch mode is otherwise automatic or the device is in the home network and the the fetch mode is automatic-home:
    1. Make a request to the system to fetch the MMS message from the URL indicated in the X-Mms-Content-Location field of the MMS notification. Once the MMS has been fetched continue with following steps.
    2. Run the *steps for filling the `MmsMessage` object with the data contained in the MMS message enclosed in the M-Retrieve.conf Protocol Data Unit*.
        1. Set the `from` of *mmsMessage* to the value of the 'From' field of the MMS message [[!MMS13]].
        2. Set the `timestamp` of *mmsMessage* to the value of the 'Date' field of the MMS message [[!MMS13]].
        3. Add a new item in the `to` array of *mmsMessage* for each of recipients in the 'To' field of the MMS message [[!MMS13]], if present.
        4. Add a new item in the `cc` array of *mmsMessage* for each of recipients in the 'Cc' field of the MMS message [[!MMS13]], if present.
        5. Add a new item in the `bcc` array of *mmsMessage* for each of recipients in the 'Bcc' field of the MMS message [[!MMS13]], if present.
        6. Set the `subject` attribute to the value of the 'Subject' field of the MMS message [[!MMS13]], if present.
        7. Set the `smil` attribute to a `DOMString` containing the SMIL object of the received MMS message [[!MMS13]], if present.
        8. For each of the media files attached to the received MMS message add a new item to the `attachments` array with:

            1. A new `Blob` object including the actual content of the attachment and the media type. The charset encoding is indicated as part of the `type` attribute of the `Blob` object by means of appending a `charset` parameter after the media type as explained in [[!RFC2046]], e.g. "text/plain; charset=utf-8".
            2. The `contentID` attribute filled with the Content-ID used to reference this attachment from the SMIL object in the incoming MMS message [[!MMS13]], if present.
            3. The `contentLocation` attribute filled with the Content-Location used to reference this attachment from the SMIL object in the incoming MMS message [[!MMS13]], if present.
        9. Set the `readReportRequested` attribute to 'true' if the 'X-Mms-Read-Report' field is present in the incoming MMS message [[!MMS13]] and has the value 'Yes', and to 'false' otherwise.
    3. Set the `state` of *mmsMessage* to 'received'.
    4. Add a new item in the `deliveryInfo` array attribute of *mmsMessage* for each unique recipient included in the 'To', 'Cc' and 'Bcc' fields (i.e. a single item if the same address has multiple ocurrences across these parameters), with the `recipient` attribute set to this recipient's address and the `deliveryStatus` attribute set to 'not-applicable'.
9. Queue a task to fire an event named `received` with the `message` attribute set to *mmsMessage*.
10. Queue a task to fire a system message named `received` of type `ReceivedMessage` with the `message` attribute set to *mmsMessage*.

The ***`fetch`*** method can be invoked to fetch an MMS message that has not been automatically fetched upon receiving the corresponding MMS notification, e.g. due to the fetch mode being manual. When this method is invoked the User Agent MUST run the following steps:

1. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
2. Return *promise* to the caller.
3. If the `messageID` parameter passed in the request matches with an MMS message that has already been fetched, or to an SMS message go to next step, otherwise let *mmsMessage* the message with `messageID` attribute equal to the `messageID` parameter and set the `state` of *mmsMessage* to 'fetching' and make a request to the system to fetch the MMS message.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Run the steps for filling the MmsMessage object with the data contained in the MMS message.

        2. Invoke *resolver*'s fulfill algorithm with *mmsMessage* as the `value` argument.

An example of manual fetch of an MMS is provided below:

```
navigator.setMessageHandler ('received', onMessageReceived)
function onMessageReceived(message) {
  if (message.type == 'sms') { window.console.log('SMS Message from ' +
                               message.from + ' received');  }
  else if (message.type == 'mms') {
    window.console.log('MMS Message from ' + message.from + ' received');
    if (message.state == 'not-downloaded') {
      window.console.log('MMS Message download started');
      navigator.messaging.mms.fetch (message.id).done(
        function(message) { window.console.log('MMS Message downloaded!');},
        function(error)   { window.console.error('Error: ' + error);} );
      }
  }
}
```

It is FFS how to report the progress of the sending and fetching of an MMS message.

Upon a delivery report of a previously sent MMS message being received, the user agent MUST

1. Let *mmsMessage* be the instance of `MmsMessage` to which this delivery report is related.
2. For each of the items in the `deliveryInfo` array attribute of *mmsMessage* whose `recipient` attribute matches one of the recipients of the MMS to which the delivery report is related,
    1. set its `deliveryStatus` attribute:
        1. to 'success', in case of successful delivery, i.e. if the value of the X-Mms-Status element in the M-Delivery.ind Protocol Data Unit [[!MMS13]] is 'Retrieved', or
        2. to 'error', in case of failed delivery, i.e. if the value of the X-Mms-Status element in the M-Delivery.ind Protocol Data Unit [[!MMS13]] is 'Expired', 'Rejected' or 'Unreachable'.
    2. set its `deliveryTimestamp` attribute to the delivery time, i.e. the 'Date' field in the M-Delivery.ind Protocol Data Unit [[!MMS13]], in case of successful delivery.
3. Queue a task to fire an event named deliverysuccess or deliveryerror respectively if the delivery was successfull or not, with
    1. the `messageID` attribute set to the `messageID` attribute of *mmsMessage*,
    2. the `serviceID` attribute set to the `serviceID` attribute of *mmsMessage*,
    3. the `recipients` attribute set to the subset of the original recipients of *mmsMessage*to which this delivery report is related, and

    4. in case of successful delivery, with each of the items in the `deliveryTimestamps` attribute set to the delivery time of the MMS message to the corresponding recipient, i.e. that in the same position of the `recipients` array.
4. Queue a task to fire a system message of type `DeliveryReport` named `deliverysuccess` or `deliveryerror` respectively if the delivery was successfull or not, with
    1. the `messageID` attribute set to the `messageID` attribute of *mmsMessage*,
    2. the `serviceID` attribute set to the `serviceID` attribute of *mmsMessage*,
    3. the `recipients` attribute set to the subset of the original recipients of *mmsMessage*to which this delivery report is related, and
    4. in case of successful delivery, with each of the items in the `deliveryTimestamps` attribute set to the delivery time of the MMS message to the corresponding recipient, i.e. that in the same position of the `recipients` array.

Upon a read report of a previously sent MMS message being received, the user agent MUST

1. Let *mmsMessage* be the instance of `MmsMessage` to which this read report is related.
2. For each of the items in the `deliveryInfo` array attribute of *mmsMessage* whose `recipient` attribute matches one of the recipients of the MMS to which the read report is related,
    1. set its `readStatus` attribute:
        1. to 'success', in case the message has been read, i.e. if the value of the X-Mms-Read-Status element in the M-Read-Orig.ind Protocol Data Unit [[!MMS13]] is 'Read', or
        2. to 'error', in case the message has been deleted without being read, i.e. if the value of the X-Mms-Status element in the M-Read-Orig.ind Protocol Data Unit [[!MMS13]] is 'Deleted without being read'.
    2. set its `readTimestamp` attribute to the read time, i.e. the 'Date' field in the M-Read-Orig.ind Protocol Data Unit [[!MMS13]], in case the message has been read.
3. Queue a task to fire an event named readsuccess or readerror respectively if the message has been read or not, with
    1. the `messageID` attribute set to the `messageID` attribute of *mmsMessage*,
    2. the `serviceID` attribute set to the `serviceID` attribute of *mmsMessage*,
    3. the `recipients` attribute set to the subset of the original recipients of *mmsMessage*to which this read report is related, and
    4. in case the message has been read, with each of the items in the `readTimestamps` attribute set to the read time of the MMS

message by the corresponding recipient, i.e. that in the same position of the `recipients` array.

4. Queue a task to fire a system message of type `ReadReport` named `readsuccess` or `readerror` respectively if the message has been read or not, with
    1. the `messageID` attribute set to the `messageID` attribute of *mmsMessage*,
    2. the `serviceID` attribute set to the `serviceID` attribute of *mmsMessage*,
    3. the `recipients` attribute set to the subset of the original recipients of *mmsMessage*to which this read report is related, and
    4. in case the message has been read, with each of the items in the `readTimestamps` attribute set to the read time of the MMS message by the corresponding recipient, i.e. that in the same position of the `recipients` array.

The `clear` method when invoked MUST run the following steps:

1. Make a request to the system to delete all the messages associated to the messaging service with identifier equal to the `serviceID` parameter.
2. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`
3. Return *promise* to the caller.
4. If an *error* occurs invoke *resolver*'s reject algorithm with *error* as the `value` argument.
5. When the request has been successfully completed:
    1. Let *serviceID* be the `serviceID` parameter passed in the request
    2. Invoke *resolver*'s fulfill algorithm with *serviceID* as the `value` argument.

H.10. EVENT HANDLERS

The following are the event handlers (and their corresponding event types) that MUST be supported as attributes by the MmsManager object.

event handler	event name	event type	short description
`onreceived`	`received`	`MessagingEvent`	handles received messages
`onsent`	`sent`	`MessagingEvent`	handles sent messages
`ondeliverysuccess`	`deliverysuccess`	`DeliveryReportEvent`	handles successful delivery reports

event handler	event name	event type	short description
`ondeliveryerror`	`deliveryerror`	`DeliveryReportEvent`	handles failure delivery reports
`onreadsuccess`	`readsuccess`	`ReadReportEvent`	handles successful read reports
`onreaderror`	`readerror`	`ReadReportEvent`	handles failure read reports
`onserviceadded`	`serviceadded`	`ServiceChangeEvent`	handle new messaging services
`onserviceremoved`	`serviceremoved`	`ServiceChangeEvent`	handle disabled messaging services

H.11. MMSSENDPARAMETERS DICTIONARY

optional DOMString serviceID
Identifier of the service through which the message is requested to be sent.

boolean requestDeliveryReport
Flag to indicate whether a delivery report is requested.

boolean requestReadReport
Flag to indicate whether a read report is requested.

H.12. SMSMESSAGE INTERFACE

The SmsMessage interface represents an SMS message as defined in [[!GSM-SMS]]. This interface is not intended to represent binary SMS, which are out of the scope of this API.

readonly attribute DOMString messageID
MUST return the identifier of the message.

readonly attribute MessageType type
MUST return the type of message, i.e. 'sms'.

readonly attribute DOMString serviceID
MUST return the messaging service id used to send / receive this message.

readonly attribute DOMString from
MUST return the sender of the message, i.e. the TP Originating Address (TP-OA) of the SMS message.

readonly attribute Date timestamp
MUST return, for received messages the time the message reached the Short Message Center, indicated in the TP-Service-Centre-Time-Stamp (TP-SCTS) parameter of the SMS message, and for sent messages the the device's date when the SMS message was sent, i.e. when the SMS-SUBMIT Protocol Data Unit was sent.

readonly attribute boolean read
MUST return 'true' if the message has been marked as read, or 'false' otherwise.

readonly attribute DOMString to
MUST return the recipient of the message, i.e. the TP Destination Address (TP-DA) of the SMS message.

readonly attribute DOMString body
MUST return text of the SMS message, i.e. the SM element contained within the TP User Data (TP-UD) field of the SMS message.

readonly attribute SmsState state
MUST return the status of the SMS message.

readonly attribute DeliveryStatus deliveryStatus
MUST return the delivery status of the SMS message.

readonly attribute Date? deliveryTimestamp
MUST return for sent messages the delivery date as reported in the TP-Discharge-Time (TP DT) parameter included in the SMS-STATUS-REPORT Protocol Data Unit or null if there is no positive knowledge about the delivery of the message. MUST return null for received messages.

readonly attribute MessageClass messageClass
MUST return the SMS message class, according to the value indicated in the TP-Data-Coding-Scheme (TP-DCS) field of the SMS.

H.13. MMSMESSAGE INTERFACE

The MmsMessage interface represents an MMS message, as defined in [[!MMS13]].

readonly attribute DOMString messageID
MUST return the identifier of the message.

readonly attribute MessageType type
MUST return the type of message, i.e. 'mms'.

readonly attribute DOMString serviceID
MUST return the messaging service id used to send / receive this message.

readonly attribute DOMString from
MUST return the sender of the message, i.e. the 'From' field of the MMS message.

readonly attribute Date timestamp
MUST return, for received messages the time the message reached the Multimedia Messaging Service Center, i.e. the 'Date' field of the MMS message, for received but not downloaded messages the timestamp of the binary SMS message used to transport the MMS notification and for sent messages the device's date when the MMS message was sent, i.e. the date when the M-Send.req Protocol Data Unit was sent by the MMS Client.

readonly attribute unsigned long? expiry
MUST return the number of seconds the message will be stored in the Multimedia Messaging Service Center and thus available for download. This time is calculated since the date the MMS Notification was sent.

readonly attribute boolean read
MUST return 'true' if the message has been marked as read, or 'false' otherwise.

readonly attribute DOMString[] to
MUST return an array containing the recipient(s) included in the 'To' field of the MMS message.

readonly attribute DOMString[] cc
MUST return an array containing the recipient(s) included in the 'Cc' field of the MMS message.

readonly attribute DOMString[] bcc
MUST return an array containing the recipient(s) included in the 'Bcc' field of the MMS message.

readonly attribute DOMString subject
MUST return the subject of the MMS message, corresponding to the 'Subject' field of the MMS message.

readonly attribute DOMString smil
MUST return the SMIL, i.e. the presentation element, unparsed as DOMString, that the messaging client needs to use to determine the way the content of the MMS message is displayed.

readonly attribute MmsAttachment[]? attachments
MUST return the set of attachments of the MMS message.

readonly attribute MmsState state
MUST return the status of the MMS message.

readonly attribute MmsDeliveryInfo[] deliveryInfo

MUST return an array with each of the items indicating the delivery status and, if applicable, the delivery time to each of the recipients of the message.

readonly attribute boolean? readReportRequested

MUST return true in case the originator of a received message requested a read report and false otherwise. MUST return null for sent messages.

H.14. MMSCONTENT DICTIONARY

DOMString subject

Indicates the subject of the MMS message, corresponding to the 'Subject' field of the MMS message.

DOMString[] to

Indicates the recipient(s) included in the 'To' field of the MMS message. There MUST be at least one recipient in any of the `to`, `cc` or `bcc` attributes.

DOMString[] cc

Indicates the recipient(s) included in the 'Cc' field of the MMS message. There MUST be at least one recipient in any of the `to`, `cc` or `bcc` attributes.

DOMString[] bcc

Indicates the recipient(s) included in the 'Bcc' field of the MMS message. There MUST be at least one recipient in any of the `to`, `cc` or `bcc` attributes.

DOMString smil

Contains the SMIL component, i.e. the presentation element that determines the way the content of the MMS message MUST be displayed.

MmsAttachment[] attachments

Contains the set of attachments of the MMS message.

H.15. MMSATTACHMENT DICTIONARY

DOMString contentID

The Content-ID parameter that MAY be used to refer to the attachment from the SMIL presentation object as described in [[!MMS13]] and [[!MIME-ENC]]. At least one of contentID and contentLocation MUST be specified if the MMS Message contains an SMIL presentation object.

DOMString contentLocation

The Content-Location parameter that MAY be used to refer to the attachment from the SMIL presentation object as described in [[!MMS13]] and [[!MIME-ENC]]. At least one of contentID and contentLocation MUST be specified if the MMS Message contains an SMIL presentation object. It may also be used as a hint to define the filename when the attachment is stored at the file system.

Blob content
The Blob object containing the media type and the content of the attachment.

H.16. MMSDELIVERYINFO DICTIONARY

DOMString recipient
The recipient of the MMS to which the delivery status is related.

DeliveryStatus deliveryStatus
The delivery status of the MMS message to a specific recipient.

Date deliveryTimestamp
The time the message was delivered to the recipient, i.e. the 'Date' field in the M-Delivery.ind Protocol Data Unit [[!MMS13]]. It is not provided if there is no positive knowledge about the delivery of the message.

ReadStatus readStatus
The read status of the MMS message to a specific recipient.

Date readTimestamp
The time the message was read by the recipient, i.e. the 'Date' field in the M-Read-Orig.ind Protocol Data Unit [[!MMS13]]. It is not provided if there is no positive knowledge about the delivery of the message.

H.17. CONVERSATION INTERFACE

The Conversation interface represents a set of messages that are grouped together either because they are exchanged among the same set of participants or because they have the same subject.

readonly attribute DOMString conversationID
MUST return the identifier of the conversation, which is globally unique within the implementation, i.e. there cannot be any other conversation with the same identifier.

readonly attribute MessageType type
MUST return the type of conversation, with value 'participants' if the conversation is defined as the set of messages exchanged among the same set of parties, and 'subject' if the conversation is defined as the set of messages with the same subject.

readonly attribute DOMString[] participants
MUST return an array containing the participants in the conversation. In case the conversation is of type 'subject' and there are messages in the conversation with a different set of participants then this attribute MUST return the union of the participants of all the messages in the conversation.

readonly attribute DOMString subject
MUST return the subject of the conversation, if there is a single one.

readonly attribute DOMString[] messageTypes
MUST return an array contining the different types of messages included in the conversation.

readonly attribute unsigned long messageCount
MUST return the number of messages in the conversation.

readonly attribute unsigned long unreadCount
MUST return the number of unread messages in the conversation.

readonly attribute DOMString lastMessageID
MUST return the identifier of the message in the conversation with the most recent timestamp.

readonly attribute MessageCursor cursor
MUST return the `MessageCursor` to access the messages in this conversation.

H.18. MESSAGINGCURSOR INTERFACE

The MessagingCursor interface allows to iterate through a list of `Conversation` elements or of messages (`SmsMessage` and/or `MmsMessage` elements).

A MessagingCursor always has, an *associated request* which is the Promise that created it.

As soon as the MessagingCursor is *accessing an element*, it MUST put its associated request in the 'processing' `readyState`, and set the `result` to null. Then, the UA MUST fetch the next/previous element asynchronously. When the element is retrieved, the associated request's `readyState` must be set to 'done' and the `result` must point to the cursor. If no element was found, the cursor's `element` property must return null.

readonly attribute any? element
This property MUST return the currently accessed element. If the cursor went past the last element or if it is currently accessing the next element, it MUST return null.

void next()
When this method is called, the cursor MUST change its internal state to accessing an element and proceed to access to the next element.
If this method is called while the cursor is in the process of accessing an element, the method MUST throw an `"InvalidStateError"` error as defined in [[!DOM4]].

void previous()
When this method is called, the cursor MUST change its internal state to accessing an element and proceed to access to the previous element.
If this method is called while the cursor is in the process of accessing

an element, the method MUST throw an `"InvalidStateError"` error as defined in [[!DOM4]].

H.19. RECEIVEDMESSAGE INTERFACE

The ReceivedMessage interface represents a system message related to a received message. This event is originated from the system and will start the application if it is not currently running.

The application that consumes this API SHOULD set a message handler for the `ReceivedMessage` system message to listen for when a system message related to a received message is fired.

readonly attribute (SmsMessage or MmsMessage) message
MUST return the `SmsMessage` or `MmsMessage` object to which this system message is related.

H.20. DELIVERYREPORT INTERFACE

The DeliveryReport interface represents a system message related to a delivery report of a sent message. This event is originated from the system and will start the application if it is not currently running.

The application that consumes this API MAY set a message handler for the `DeliveryReport` system message to listen for when a system message related to a received delivery report is fired.

readonly attribute DOMString serviceID
MUST return the identifier of the service used to send the message to which this delivery report is related.

readonly attribute DOMString messageID
MUST return the identifier of the message to which this delivery report is related.

readonly attribute DOMString[] recipients
MUST return an array containing the addresses of the subset of the original recipients of the message to which this delivery report is related. As delivery reports related to just part of the recipients of the MMS message are possible, this array may not contain the full list of recipients to which the MMS message was sent. If the delivery report is related to an SMS message then the array will contain a single item corresponding to the single recipient of the SMS message.

readonly attribute Date[]? deliveryTimestamps
MUST return an array containing the delivery dates for each of the recipients to which this delivery report event relates. Each element in the array refers to the recipient in the same position of the `recipients` array. It MUST return null if case of delivery failure (e.g. message expired)

H.21. READREPORT INTERFACE

The ReadReport interface represents a system message related to a read report of a sent MMS message. This event is originated from the system and will start the application if it is not currently running.

The application that consumes this API MAY set a message handler for the `ReadReport` system message to listen for when a system message related to a received read report is fired.

readonly attribute DOMString serviceID
MUST return the identifier of the service used to send the message to which this read report is related.

readonly attribute DOMString messageID
MUST return the identifier of the message to which this read report is related.

readonly attribute DOMString[] recipients
MUST return an array containing the addresses of the subset of the original recipients of the message to which this read report is related. As read reports related to just part of the recipients of the MMS message are possible, this array may not contain the full list of recipients to which the MMS message was sent.

readonly attribute Date[]? readTimestamps
MUST return an array containing the read dates by each of the recipients to which this read report event relates. Each element in the array refers to the recipient in the same position of the `recipients` array.

H.22. MESSAGINGEVENT INTERFACE

The MessagingEvent interface represents events related to a message sent or received.

readonly attribute (SmsMessage or MmsMessage) message
MUST return the `SmsMessage` or `MmsMessage` object to which this event is related.

H.23. DELIVERYREPORTEVENT INTERFACE

The DeliveryReportEvent interface represents events related to a delivery report of a sent message.

readonly attribute DOMString serviceID
MUST return the identifier of the service used to send the message to which this delivery report event is related.

readonly attribute DOMString messageID
MUST return the identifier of the message to which this delivery report event is related.

readonly attribute DOMString[] recipients
MUST return an array containing the addresses of the subset of the original recipients of the message to which this delivery report event is related. As delivery reports related to just part of the recipients of the MMS message are possible, this array may not contain the full list of recipients to which the MMS message was sent. If the delivery report is related to an SMS message then the array will contain a single item corresponding to the single recipient of the SMS message.

readonly attribute Date[]? deliveryTimestamps
MUST return an array containing the delivery dates for each of the recipients to which this delivery report event relates. Each element in the array refers to the recipient in the same position of the `recipients` array. It MUST return null if case of delivery failure (e.g. message expired)

H.24. READREPORTEVENT INTERFACE

The ReadReportEvent interface represents events related to a read report of a sent message.

readonly attribute DOMString serviceID
MUST return the identifier of the service used to send the message to which this read report event is related.

readonly attribute DOMString messageID
MUST return the identifier of the message to which this read report event is related.

readonly attribute DOMString[] recipients
MUST return an array containing the addresses of the subset of the original recipients of the message to which this read report event is related. As read reports related to just part of the recipients of the MMS message are possible, this array may not contain the full list of recipients to which the MMS message was sent.

readonly attribute Date[]? readTimestamps
MUST return an array containing the read dates for each of the recipients to which this read report event relates. Each element in the array refers to the recipient in the same position of the `recipients` array.

H.25. SERVICECHANGEEVENT INTERFACE

The ServiceChangeEvent interface represents events related to messaging services enabled or disabled.

readonly attribute DOMString serviceID
MUST return the identifier of the messaging service which is enabled or disabled.

H.26. MESSAGINGFILTER DICTIONARY

The MessagingFilter Dictionary represents a filter that is used to select a set of messages (e.g. to be provided upon invoking the `findMessages` or the `findConversations` method in the `Messaging` interface).

MessageType type
Indicates whether just the SMS or MMS messages are to be provided in the results of this filter, respectively if it is set to string value 'SMS' or 'MMS'.

Date startDate
Indicates that messages with timestamp previous to this date will not be provided in the results of this filter.

Date endDate
Indicates that messages with timestamp after this date will not be provided in the results of this filter.

DOMString from
Indicates that just messages sent from this number are to be provided in the results of this filter.

sequence<DOMString> recipients
Indicates that just messages sent to one of these numbers are to be provided in the results of this filter.

(SmsState or MmsState) state
Indicates whether the results of this filter just needs to return the messages matching the indicated state.

DOMString serviceID
Indicates that just messages associated to the messaging service with this identifier are to be provided in the results of this filter.

boolean read
Indicates whether just read or unread messages are to be provided in the results of this filter, respectively if it is set to 'true' or 'false'.

H.27. FILTEROPTIONS DICTIONARY

DOMString sortBy
Indicates the attribute on which the filtered messages are sorted.

DOMString sortOrder
Indicates the order on which the filtered messages are sorted with possible values 'ascending' and 'descending'.

unsigned long limit
  Indicates the maximum number of messages that can be returned as a result of applying the corresponding filter.

H.28. ENUMERATIONS

The attibute `type` can have the following values:

sms
  Corresponding to SMS message(s).

mms
  Corresponding to MMS message(s).

The attibute `messageClass` in an `SmsMessage` can have the following values:

class-0
  The message is of class 0.

class-1
  The message is of class 1.

class-2
  The message is of class 2.

class-3
  The message is of class 3.

normal
  The message is of class 1 (same as 'class-1').

The attibute `state` in an `SmsMessage` can have the following values:

received
  The message is an inbound message.

sending
  The message is in process of being sent.

sent
  The message has been successfully sent.

failed
  The message is an outbound message whose submission has failed.

The attibute `state` in an `MmsMessage` can have the following values:

not-downloaded
  The message is an inbound message, for which an MMS notification has been received but that has not yet been downloaded.

fetching
: The message is an inbound message, for which an MMS notification has been received, and whose download has started and not yet finished.

received
: The message is an inbound message.

sending
: The message is in process of being sent.

sent
: The message has been successfully sent.

failed
: The message is an outbound message whose submission has failed.

The attibute `deliveryStatus` can have the following values:

success
: The message has been succesfully delivered to the recipient.

pending
: The message is pending delivery.

error
: The delivery of the message has failed.

not-applicable
: The delivery status is not applicable either because a delivery report has not been requested or because the message is an inbound message

The attibute `readStatus` can have the following values:

success
: The message has been read by the recipient.

pending
: There is no positive knowledge that the message has been read by the recipient.

error
: The delivery of the message has failed.

not-applicable
: The read status is not applicable either because a read report has not been requested or because the message is an inbound message

The MMS fetch mode can have the following values:

automatic
: MMS message is fecthed right after the reception of the MMS notification.

automatic-home
MMS message is fecthed right after the reception of the MMS notification if the device is located in the home network, i.e. not roaming, otherwise the message will be fetched upon the device entering in the home network again or the user manually requesting it.

manual
The message is not fecthed until the user manually requests it.

never
MMS message retrieval is completely disabled.

H.29. ACKNOWLEDGEMENTS

The editors would like to express their gratitude to the Mozilla B2G Team and specially to Jonas Sicking, Mounir Lamouri and Vicamo Yang for their technical guidance, implementation work and support.

I. TELEPHONY API

W3C Editor's Draft 14 October 2014

This version:
http://www.w3.org/2012/sysapps/telephony/

Latest published version:
http://www.w3.org/TR/telephony/

Latest editor's draft:
http://www.w3.org/2012/sysapps/telephony/

Editors:
Marcos Cáceres, Mozilla
José M. Cantera, Telefónica
Eduardo Fullea, Telefonica
Zoltan Kis, Intel
John Lyle, University of Oxford

This specification defines an API to manage telephone calls. A typical use case of the *Web Telephony API* is the implementation of a 'Dialer' application supporting multiparty calls and multiple telephony services.

Implementors should be aware that this specification is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation stage should join the aforementioned mailing lists and take part in the discussions.

Significant changes to this document since last publication are documented in the Changes section.

I.1. INTRODUCTION

The *Web Telephony API* allows applications to manage interaction with telephony call signaling, but does not handle audio channels management.

An example of making a telephony call is provided below:

```
    var telCall = navigator.telephony.dial('+1234567890');

    telCall.onactive = function(e) {
        window.console.log('Connected!');
    }

    telCall.ondisconnected = function(e) {
        window.console.log('Disconnected!');
        // update call history
    }

    telCall.onerror = function(e) {
        window.console.error(e);
    }
```

The use cases for this specification are collected in the wiki page of this API.

The following specifications informed the design of the *Web Telephony API*: for GSM the [[!GSM-CALL]] suite, for IMS/SIP the [[!IMS]] suite, for XMPP the [[!JINGLE]] specification. Note, however, that IMS/SIP and XMPP are not supported in this version.

It is likely that the same API would work also for SIP and XMPP calls with the exception of multiparty call handling, which is modeled after the cellular multiparty calls. Future versions of this specification will probably add SIP and XMPP conference support.

It is under discussion whether a system message should be propagated when a CDMA telephony call is active, since not all CDMA networks support concurrent services. Therefore, many applications will lose their data connection when the end user is in a voice call.

This specification defines conformance criteria that apply to a single product: the *user agent* that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [[!WEBIDL]], as this specification uses that specification and terminology.

I.2. DEPENDENCIES

This specification depends on the following interfaces and concepts defined in other specifications.

The following dependencies are defined in [[!HTML]]: *`EventHandler`* interface, *queue a task*, *event handler*, *origin*, *task source*.

The following dependencies are defined in [[!DOM4]]: the *`Event`* and the *`Promise`* interfaces, the concepts of a *resolver*, *fire an event*.

I.3. TELEPHONY SERVICES

A *telephony service* manages telephony operations associated with a subscriber identity, which is registered with a telephony service provider. For example, in cellular telephony, a telephony service is associated with SIM card (Subscriber Identity Module). A telephony service can use different protocols for telephony signaling and media (e.g. GSM, CDMA, VoLTE, etc.) with the same identity.

Each telephony service has a unique *telephony service id*, which identifies a telephony service together with a user identity in the system. For telephony services that make use of a SIM card, it is RECOMMEDED that the ICC-ID be used for the service identifier.

It is strongly RECOMMENDED a implementations **do not** use use the MSISDN as the telephony service id. The MSISDN cannot guarantee uniqueness.

A user agent can access zero or more telephony services. Which telephony services are available to the user agent is dictated by policy, or by the user choosing which services are available to the user agent. When one or more telephony services is available to the user agent, one serves as the default telephony service.

In the API, telephony services are represented by a DOMString that maps to a telephony service id for each telephony service available to an origin.

Access to a telephony service by any origin is restricted by a security policy. See the security and privacy considerations section for more details.

A telephony service observes the call states of a telephony call in the supported telephony backend(s) and networks and reflects those changes in the API through [[!DOM4]] events and promises.

I.3.1. Changing telephony services

Telephony services can be added or removed from the system at any time (e.g., the user pops out the SIM card or adds a different SIM card; the user tells the system to only allow certain applications to access a particular telephony service, etc.).

When a telephony service is either added or removed in the system, the user agent MUST run the steps to change the telephony service:

1. Let *name* be `serviceremoved` if the telephony service was removed, or `serviceadded` otherwise.
2. Let *event* be a new TelephonyServiceEvent, with *name* as the event name, which does not bubble, is not cancelable, has no default action, and whose serviceId attribute is set to the telephony service id of the telephony service that initiated this algorithm.
3. If the service has been removed and it's the default telephony service for the origin, then change default telephony service.
4. Queue a task to fire an event *event* at the TelephonyManager.

I.3.2. Default Telephony Service

The *default telephony service* is the telephony service that serves as default for telephony operations for the origin of an application. The default telephony service can be changed by the user, and if supported, through the changeDefaultService() method of the TelephonyManager object.

If there is no default telephony service for the origin, the user agent SHOULD use the system's default telephony service, if available and if allowed by policy. If there are no telephony services available to use as the default for the origin, the default telephony service is null.

When there is a need to *changes default telephony service*, the user agent MUST:

1. Let *service id* be the telephony service id of the new default telephony service, or `null` otherwise.
2. Let *event* be a new TelephonyServiceEvent with the event name `defaultchanged`, which does not bubble, is not cancelable, has no default action, and whose serviceId attribute to *service id*.
3. Queue a task to change the defaultServiceId attribute of the TelephonyManager object to *sercive id*, and fire *event* at the TelephonyManager.

I.3.3. Telephony Calls

A *telephony call* results from a telephony service's attempt to establish a connection for communication between two or more parties. Telepony calls involving more than two parties are referred to as a multiparty call.

In the process of establishing and maintaining a connection between multiple parties, a telephony call transitions through various call states. A telephony call is always in some call state, which can change over time.

Telephony calls initiated by a telephony service of the system is an *outbound call*. Conversely, a telephony calls from a remote party is an *inbound call*.

Every telephony call has a *call id*, which is a string that uniquely identifies the call and call history.

An *active call* is a Call in the active state representing a connected call which is bound to the media input and output devices (e.g. microphone, speaker, tone generator). Note that a call on hold is also active from a call signaling point of view, but not bound to media input and output devices.

Whenever a call is added to or removed from the calls array, the user agent MUST queue a task to fire an event named `callschanged`.

Within the API, a telephony call is represented by the Call typedef - which encapsulates both objects that implement the TeleponyCall interface and the ConferenceCall interface.

I.3.4. `Call` typedef

I.3.5. Multiparty calls

A *multiparty call* (also commonly referred to as a *conference call*) is a telephony call with multiple remote party participants, which is controlled as a single telephony call, i.e. it can be held, activated, disconnected from all participants. Other calls can be joined with a multiparty call.

Every multiparty call has a unique *conference id* that identifies a multiparty call in the system.

A *remote party id* uniquely identifies a participant (a.k.a. remote party) in a telephony call in the given telephony service, such as a phone number.

This of the API version supports GSM multiparty calls and CDMA 3-way calls.

In the API, a multiparty call is represented by any object that implements the ConferenceCall interface.

The following needs to be described in an algorithm: For multiparty calls, the implementation MUST generate a unique identifier stored in the `callId` attribute. In GSM, the `callId` of any participating call could be used in a multiparty operation, and the telephony network will reply with using the same value. But for forward compatibility reasons, the implementation MUST generate a separate `callId` value for the multiparty call, which MUST be unique in the system and the local call history. For GSM multiparty functionality, The implementation MUST map this to an appropriate transaction identifier accepted by the telephony service. Also, the transaction identifiers received from the network MUST be mapped back by the implementation to the unique conference call identifier of the multiparty call.

I.4. CALL STATES

A *call state* represents the state of interaction between a telephony service, the telephony network, the API, and one or more remote parties.

Within the API, the call states are represented by the `CallState` enum.

The call states referenced in this API are:

dialing
An outbound call is being dialed by a telephony service.

connecting
A request to establish the call has been made and it is progressing.

alerting
The destination number has been reached and alerting is taking place.

active
The call is ongoing.

incoming
An incoming call is being received whilst no other call is progressing.

waiting
An incoming call that has been received whilst there was another call progressing, and the call waiting service is active.

accepted
An incoming call has been accepted and is being connected.

holding
The call is being put on hold.

held
The call has been put on hold.

resuming
The call, which was on hold, is being resumed.

redirecting
The call is being redirected to another remote party from the same telephony service.

transferring
The call is being transferred to another remote party from the same telephony service.

disconnecting
A request to disconnect the call has been made and it is progressing.

disconnected
The call has been disconnected and this object is invalid for call control.

joining
The call is being joined with another call to become a multiparty call.

multiparty
The call is a multiparty call.

splitting
The call is being split from a multiparty call.

I.4.1. State changes

Whenever there is a change in the state attribute the user agent MUST:

1. Queue a task to fire an event named `statechange` with the new value of the state attribute.

Since call state transitions depend on protocol, network equipment, modem, etc., the implementation MUST NOT fire error events on assumed erroneous state transitions. MUST always re-synchronize any eventual internal states to the current call state reported by the telephony system. The implementation MUST NOT set the call state to any other value than specified in the descriptions of the methods of this interface.

Since call states can differ depending on the protocol, do we need to expose the service and the protocol used for making the call? See issue 125.

Note that compliant implementations may not be reporting such events when they occur (e.g. OTA SIM update or hot-swappable SIM cards). See issue 127.

I.4.2. `CallState` enum

```
enum CallState {
    "dialing",
    "connecting",
    "alerting",
    "active",
    "incoming",
    "waiting",
    "accepted",
    "holding",
    "held",
    "resuming",
    "redirecting",
    "transferring",
    "disconnecting",
    "disconnected",
    "joining",
    "multiparty",
    "splitting"
};
```

Some of these states are *soft states*, that is, transitory states in which the application is placed after making a request to the telephony system and until it is completed. For instance the application remains in "holding" state since it invokes the hold() method and until the telephony system actually holds the call. In some implementations these soft states can be skipped. The following are the soft states defined by this specification: "accepted", "disconnecting", "holding", "resuming".

On the contrary, *hard states* MUST be supported by the implementation: "dialing", "alerting", "active", "disconnected", "incoming", "waiting", "held". For calls participating in multiparty calls, the following additional call states MUST be supported: "joining", "splitting", "multiparty". For call transfer functionality, the additional "transferring" state MUST be supported.

I.4.6. Receiving calls (inbound states)

The device can receive phone calls from any active telephony service, even simultaneously, in which case the user agent arbitrates the calls either by a policy, or by the user by choosing which call to accept.

For call setup on received calls, the following call states MUST be supported in this order:

1. "incoming"/"waiting".
2. "accepted".
3. "active".

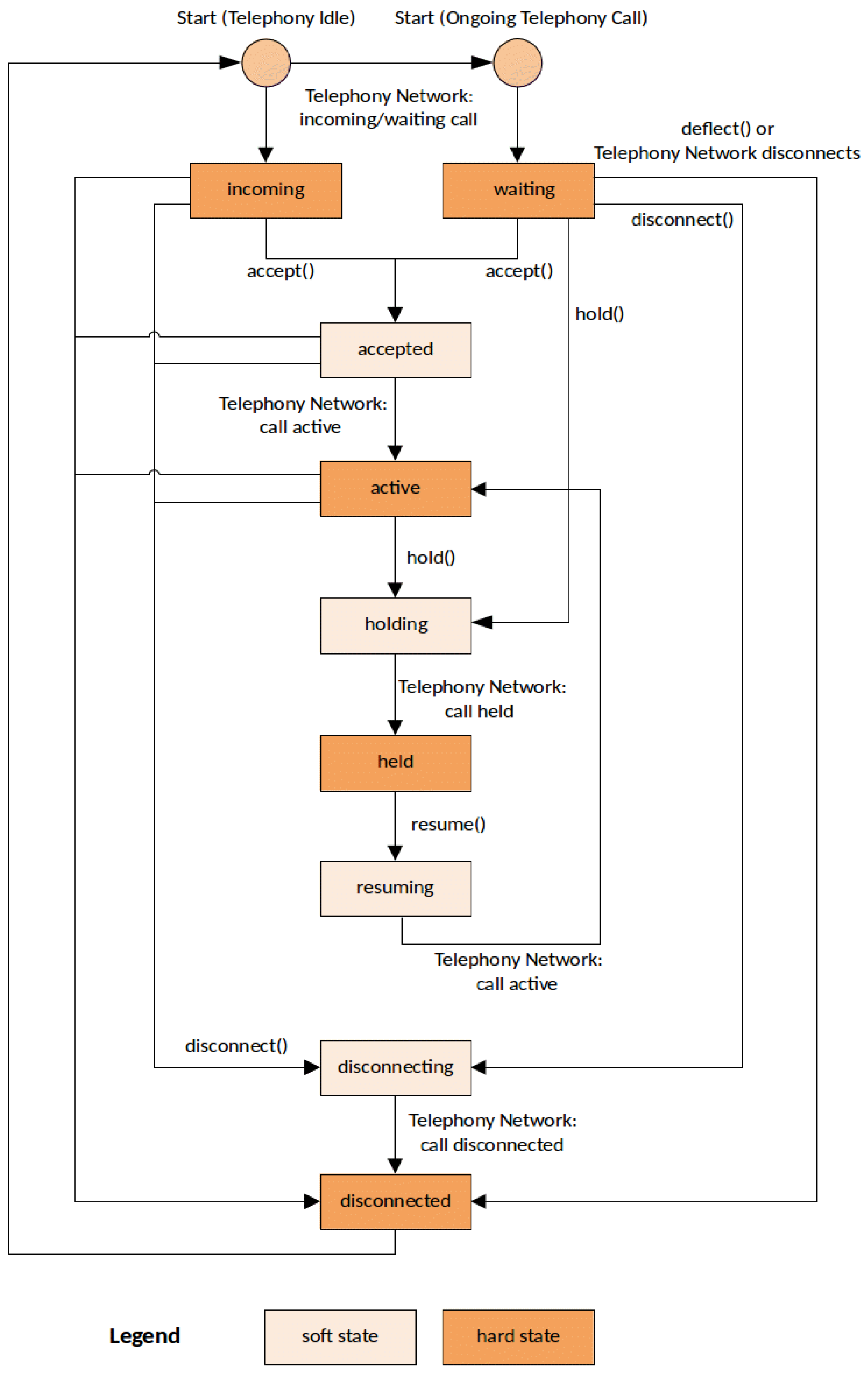
On received calls, telephony protocols also use a "ringing" state, set by the mobile terminal when local call alerting starts, in order to notify the remote party about the ongoing alerting (ringing can actually be e.g. a beep, ring tone, or vibration pattern). This is considered to be responsibility of implementations: if the modem expects this state to be set, implementations

MUST make sure to set it. Dialer applications are not expected to set this state in the current version of the specification.

CDMA cannot report all these states in the expected sequence. The 'connected' state (i.e. when the mobile station send the Service Connect Completion Message or Connect Order, depending on whether the call is mobile originated or terminated) is immediately followed by voice media transmission – there is transition to 'active' before the media arrives. Therefore it should be up to the implementation to determine which events to fire and in which order. Dialer applications need to be prepared to handle such cases.

Upon a new incoming or waiting call, the user agent MUST execute the following steps:

1. Let *incomingCall* be a new instance of TelephonyCall.
2. Set the `state` of *incomingCall* to "incoming" in case there is no other call in `active` state, or otherwise set it to "waiting".
3. Add *incomingCall* to the calls array.
4. Queue a task to fire an event named `statechange` at the *incomingCall* object.
5. Queue a task to fire an event named `incoming` at the TelephonyManager object managing the call.
6. Queue a task to fire an event named `callschanged` at the TelephonyManager object managing the call.


Start (Telephony Idle)
Start (Ongoing Telephony Call)
Telephony Network: incoming/waiting call
deflect() or Telephony Network disconnects
incoming
waiting
disconnect()
accept()
accept()
hold()
accepted
Telephony Network: call active
active
hold()
holding
Telephony Network: call held
held
resume()
resuming
Telephony Network: call active
disconnect()
disconnecting
Telephony Network: call disconnected
disconnected
Legend
soft state
hard state

The figure depicts the most usual state transitions for received calls.

I.4.7. Making calls (outbound states)

The following needs to be redefined algorithmically in terms of the telephony service.

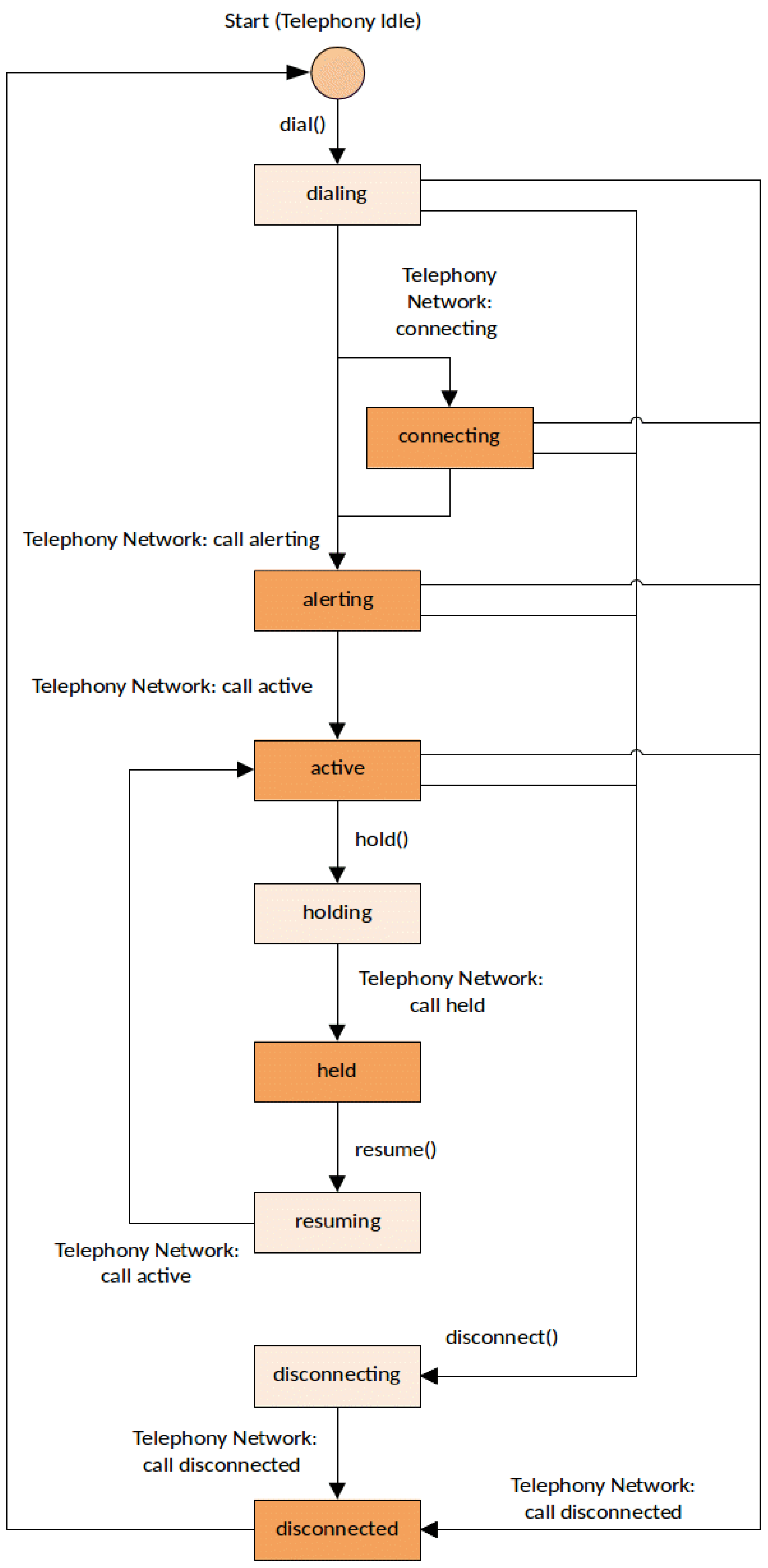
To *make a call* with a *remote party identifier*, optionally a *telephony service*, and optionally a *hide caller id*, the user agent MUST ...

Even if there is no SIM card and no other telephony services are available, but emergency calls are known to be possible (e.g. because a cellular modem is present), it is considered as a default service with only emergency call capability, and the implementation MUST define a telephony service id for it. When not even emergency calls are possible (e.g. it is a purely IP based implementation and there is no cellular modem), the implementation MAY use empty string for default telephony service id, but it is encouraged that a default service is created, with the methods throwing a `NotSupported` error.

For making a call, the telephony service transitions through the following states in order. Errors can occur at each state, which can result in the call becoming disconnected:

1. dialing
2. alerting
3. active

On the telephony services which support the "connecting" call state (e.g. GSM and CDMA, for call routing, forwarding, voicemail handling etc), implementations SHOULD support this state too, between the "dialing" and "alerting" states. Dialer applications can associate the protocol with the telephony service used for the call.


Start (Telephony Idle)
dial()
dialing
Telephony Network: connecting
connecting
Telephony Network: call alerting
alerting
Telephony Network: call active
active
hold()
holding
Telephony Network: call held
held
resume()
resuming
Telephony Network: call active
disconnect()
disconnecting
Telephony Network: call disconnected
Telephony Network: call disconnected
disconnected

The figure depicts the most usual state transitions for dialed calls.

I.5. DISCONNECTING CALLS

The reasons why a call can become disconnected are as follows. Within the API the following disconnection reasons are represented by the the `DisconnectReason` enum.

local
The call was disconnected by the user, or the device, and no more specific reason is known.

remote
The call was disconnected by the remote party, and no more specific reason is known.

network
The call was disconnected by the network, and no more specific reason is known.

busy
The call was disconnected by the network, because the remote party was busy.

rejected
The call was disconnected because the remote party rejected the call.

redirected
The call has been redirected to another subscriber.

unreachable
The call was disconnected by the network, because the remote party was unreachable by the network.

no-answer
The call was disconnected by the network, because the remote party has not answered and the call has timed out.

network-unreachable
The call was disconnected because the network was unreachable.

barred
The call was disconnected because it was barred.

no-service
The call was not made because there is no telephony service set up and enabled (e.g. no SIM card).

invalid-number
The call was disconnected by the network, because the remote party identifier was invalid.

When a telephony service is notified of call disconnection of the Call object *telCall*, it MUST run the steps to disconnect:

1. Queue a task to:
    1. Remove the *telCall* object from the calls array.
    2. set the `state` of *telCall* to "disconnected".
    3. fire an event named `statechange` at the *telCall* object.
    4. fire an event named `disconnected` at the *telCall* object.

The `param` attribute of the `disconnected` event MUST be set to the DisconnectReason, if available, or otherwise it MUST be set to `null`. At least the following values MUST be supported for the disconnect reason: "local", "remote" and "network". The rest of the DisconnectReason values SHOULD be supported.

I.5.1. `DisconnectReason` enum

```
enum DisconnectReason {
    "local",
    "remote",
    "network",
    "busy",
    "rejected",
    "redirected",
    "unreachable",
    "no-answer",
    "network-unreachable",
    "barred",
    "no-service",
    "invalid-number"
};
```

I.6. TASK SOURCE

The task source for all tasks queued in this specification is the *telephony task source*.

I.7. EXTENSIONS TO `NAVIGATOR` OBJECT

The TelephonyManager interface is exposed on [[!HTML]]'s `Navigator` object.

readonly attribute TelephonyManager telephony

I.7.1. The `telephony` attribute

When getting the *telephony* attribute, the user agent MUST return the TelephonyManager object, which provides the ability to interface with the telephony service of the device.

I.8. TELEPHONYMANAGER INTERFACE

The TelephonyManager interface provides access to telephony functionality, and manages the lifecycle of the Call objects.

readonly attribute Call? activeCall
readonly attribute Call[] calls
readonly attribute DOMString[] emergencyNumbers
readonly attribute DOMString[] serviceIds
readonly attribute DOMString? defaultServiceId
Promise changeDefaultService(DOMString serviceId)
TelephonyCall dial (DOMString remoteParty, optional DialOptions options)
Promise sendTones(DOMString tones, optional ToneOptions options)
Promise startTone(DOMString tone, optional ToneOptions options)
Promise stopTone(optional DOMString serviceId)
attribute EventHandler onincoming
attribute EventHandler oncallschanged
attribute EventHandler onserviceadded
attribute EventHandler onserviceremoved
attribute EventHandler ondefaultchanged

I.8.1. The `activeCall` attribute

When getting the *activeCall* attribute, the user agent MUST return the Call object that represents the active call. If there is no active Call return `null`.

I.8.2. The `calls` attribute

When getting the *calls* attribute, the user agent MUST return an array, which can be empty, of Call objects managed by this objects.

TelephonyCall objects belonging to a multiparty call are managed by the corresponding ConferenceCall object and won't be present in this array.

I.8.3. The `emergencyNumbers` attribute

When getting the *emergencyNumbers* attribute, the user agent MUST return an array, which can be empty, of telephone numbers for the emergency services in the current geographical area.

I.8.4. The `serviceIds` attribute

When getting the *serviceIds* attribute, the user agent MUST return an array, which can be empty, of telephony service id's.

I.8.5. The `defaultServiceId` attribute

When getting the *defaultServiceId* attribute, the user agent MUST return the DOMString that represents the id of the default telephony service (if any). Otherwise, it returns `null`.

I.8.6. The `sendTones()` method

The *sendTones()* method requests a telephony service emit one or more [[!DTMF]] tones. When invoked, the user agent MUST run the following steps:

1. If the ToneOptions parameter specifies the serviceId to be used, then validate and use that value, otherwise use the default telephony service for sending the tones.
2. If the ToneOptions parameter specifies the tone `duration`, then validate and use that value, otherwise use a default value.
3. If the ToneOptions parameter specifies the tone `gap`, then validate and use that value, otherwise use a default value.
4. Request from the telephony system to send the specified tones.
5. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`.
6. Return *promise* to the caller and continue the following steps asynchronously.
7. If the request to the telephony system is successful, or if the telephony system does not support feedback about the result of the request, invoke *resolver*'s `accept()` method with no arguments.
8. If the request to the telephony system is unsuccessful, invoke *resolver*'s `reject()` method, with no arguments.

I.8.7. Tones

Tone value can be any of the following characters: 0-9; A-D; *; `#`.

The above needs to be converted to ABNF

I.8.8. The `stopTone()` method

The *stopTone()* method stops emitting a [[!DTMF]] tone in the default or the specified telephony service. When invoked, the user agent MUST run the following steps:

1. If the platform does not support long press [[!DTMF]] tones, throw a `NotSupported` error.
2. If the provided parameters are invalid, or there is no tone playing on the specified telephony service, throw an `InvalidStateError` error.
3. Otherwise, request from the telephony system to stop sending the specified tone.
4. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`.

5. Return *promise* to the caller and continue the following steps asynchronously.
6. If the request to the telephony system is successful, or if the telephony system does not support feedback about the result of the request, invoke *resolver*'s `accept()` method with no arguments.
7. If the request to the telephony system is unsuccessful, invoke *resolver*'s `reject()` method, with no arguments.

I.8.9. The `startTone()` method

The *startTone()* method starts emitting a [[!DTMF]] tone with the platform default or specified delay, in the platform default or the specified telephony service. A `Promise` object will be returned in order to notify the result of the request.

When the `startTone` method is invoked, the user agent MUST run the following steps:

1. If the platform does not support long press [[!DTMF]] tones, throw a `NotSupported` error and finish these steps. In this case applications may then use the `sendTones` method for sending [[!DTMF]].
2. If the ToneOptions parameter specifies the serviceId to be used, then validate and use that value, otherwise use the default telephony service for sending the tones.
3. If the ToneOptions parameter specifies the tone `duration`, then ignore that value.
4. If the ToneOptions parameter specifies the tone `gap`, meaning the delay before sending the tone, then validate and use that value, otherwise use a default value.
5. Request from the telephony system to start sending the specified tone. The tone SHOULD play until the `stopTone` method is called.
6. Let *promise* be a new `Promise` object and *resolver* its associated `resolver`.
7. Return *promise* to the caller and continue the following steps asynchronously.
8. If the request to the telephony system is successful, or if the telephony system does not support feedback about the result of the request, invoke *resolver*'s `accept()` method with no arguments.
9. If the request to the telephony system is unsuccessful, invoke *resolver*'s `reject()` method, with no arguments.

I.8.10. The `dial()` method

The *dial(remoteParty, options)* method initiates a new telephony call. When invoked the user agent MUST run the following steps:

Note that verification of the format of the *remoteParty* argument is left to the telephony service. Providing an *remoteParty* in the invalid format will generally result in the call disconnecting because the telephony service or telephony network will deem it an invalid number.

1. Let *remote party* be the value of the value of the *remoteParty* argument.
2. Let *service* initially be either the default telephony service, if one is available to the document's effective script origin, or null otherwise.
3. If the *options* argument is used, and it has a serviceId member:
    1. If the value of serviceId member exactly matches the telephony service id accessible to the document's effective script origin, then let *service* be the telephony service that matches that value.
    2. If the serviceId member is not an empty string and that does not match any telephony service id available to the document, throw a "NotFoundError" exception and terminate these steps.
4. If the *options* argument is used, and it has a hideCallerId member, then let *hide caller id* be the value of the hideCallerId member.
5. Let *telCall* be a new instance of TelephonyCall.
6. Set *telCall*'s remoteParty attribute to *remote party id*.
7. Return *telCall* and continue asynchronously.
8. Make a call using *remote party*, *service* and, and, if it was defined, *hide caller id*.

I.8.11. The `changeDefaultService()` method

The *changeDefaultService()* method provides a means to change the default telephony service used by the user agent. When invoked, the user agent runs the the following steps:

1. Let *potential service* be the first argument passed to this operation.
2. Let *promise* be a new `Promise` object and *resolver* its associated resolver.
3. Return *promise* and continue the following steps asynchronously.
4. If *potential service* does not exactly match the identifier of any telephony service known to the user agent, run the following sub-steps and terminate this algorithm:
    1. Let *error* be a new DOMError object whose name is "NotFoundError".
    2. Invoke *resolver*'s reject(value) method with *error* as the value argument.
5. If *potential service* exactly matches the service id of the current default telephony service, run the following sub-steps and terminate this algorithm:
    1. Invoke *resolver*'s accept(value) method with *potential service* as the value argument.
6. Otherwise, run the steps to change the default service, with *potential service* as the telephony service id, and *promise* as the `Promise`.

The *steps to change the default service* are given by the following algorithm. This abstract operation takes as an argument a telephony service id and an optional `Promise`.

1. Make a platform/system specific request to the underlying system to change from the current default telephony service to the one identified by *service id*.
2. Possibly wait indefinitely.
3. If it's not possible (for whatever reason: timeout, security, etc.) to change the default telephony service, and if *promise* was passed, run the following sub steps and terminate this algorithm:
    1. Let *error* be a new DOMError object whose name is "NoModificationAllowedError".
    2. Call *resolver*'s reject(value) method with *error* as the value argument.
4. Otherwise, queue a task to:
    1. Change the defaultServiceId attribute to the telephony service id of the new default telephony service.
    2. If `Promise` was passed, invoke *resolver*'s accept(value) method with the id of the new default service as value argument.
    3. Fire an event named `servicechange` at the telephony attribute of the navigator object.

I.9. EVENT HANDLERS

The following are the event handlers are implemented by the TelephonyManager interface.

event handler	**event name**	**event type**	**short description**
`onincoming`	`incoming`	TelephonyEvent	handles incoming and waiting calls.
`oncallschanged`	*`callschanged`*	`Event`	handles change in the calls array.
`onserviceadded`	*`serviceadded`*	TelephonyServiceEvent	handles a new enabled telephony service.
`onserviceremoved`	*`serviceremoved`*	TelephonyServiceEvent	handles a disabled telephony service.
`ondefaultchanged`	*`defaultchanged`*	TelephonyServiceEvent	handles the change of default telephony service.

I.9.12. DialOptions Dictionary

boolean hideCallerId
DOMString serviceId

I.9.13. The `hideCallerId` member

The *hideCallerId* member represents whether the local party identifier is to be hidden or displayed to the remote party being called. If missing, the user agent uses the default configuration for the telephony service that initiated the call.

Only some protocols support hidding the identity of the local party when making a call.

I.9.14. The `serviceId` member

The *serviceId* member represents the telephony service id of the telephony service to be used when dialing.

I.9.15. ToneOptions Dictionary

unsigned long duration
unsigned long gap
DOMString serviceId

I.9.16. The `duration` member

The *duration* member represents the duration (mark) in milliseconds of the [[!DTMF]] tones to be sent.

I.9.17. The `gap` member

The *gap* member represents the duration in milliseconds of the time gap (space) before a [[!DTMF]] tone.

I.9.18. The `serviceId` member

The *serviceId* member represents the telephony service id of the telephony service to be used when dialing.

I.10. TELEPHONYEVENT INTERFACE

Defines telephony events for TelephonyCall state changes, including handling incoming and waiting calls.

readonly attribute Call call

I.10.1. The `call` attribute

When getting the *call* attribute, the user agent MUST return the TelephonyCall that triggered the event.

I.11. TELEPHONYSERVICEEVENT INTERFACE

Defines a telephony event for notifying a changed telephony service.

readonly attribute DOMString serviceId

I.11.1. The `serviceId` attribute

When getting the *serviceId*, the user agent MUST return the telephony service id of the telephony service that triggered the event.

I.11.2. `TelephonyServiceEventInit` dictionary

DOMString serviceId

I.11.3. The `serviceId` member

The *serviceId* member represents the telephony service id of a telephony service.

I.12. CALLHANDLER INTERFACE

The `CallHandler` interface provides common properties and event handling infrastructure that is implemented by other interfaces in this specification, e.g. TelephonyCall and ConferenceCall. It serves as an editorial aid in this specification, and has no functional utility on its own.

void resume()
void hold()
void disconnect()
readonly attribute DOMString callId
readonly attribute DOMString serviceId
readonly attribute CallState state
attribute EventHandler onerror
attribute EventHandler onstatechange

I.12.1. The `resume()` method

The *resume()* method requests the telephony system to resume resume a held Call. When invoked, the user agent MUST run the following steps:

1. If `state` is not equal to "held", then throw an `InvalidStateError` exception.
2. Otherwise make a request to the telephony system to resume the call. If the request is acknowledged, then set `state` to "resuming".

I.12.2. The `hold()` method

The *hold()* method requests the telephony system put the Call on hold. When invoked, the user agent MUST run the following steps:

1. If `state` is not equal to `active`, then an throw and `InvalidStateError` exception.
2. Otherwise make a request to the telephony system to hold the call.
3. Return, and continue these steps asynchronously.
4. If the request is acknowledged then set `state` to "holding".

I.12.3. The `disconnect` method

The *disconnect* method, if invoked on a TelephonyCall, it initiates releasing of the telephony call. If invoked on a ConferenceCall, it initiates releasing the multiparty call, and each participating TelephonyCall object.

If the telephony service is CDMA, throw "`NotSupported`" DOMError.

Depending on the protocol, there may be restrictions on methods. For instance, GSM does not permit disconnecting a held call. Also, disconnecting a participant in a held multiparty call is not supported. Also, if the controlling party disconnects in IS-41 3-way call in CDMA, then all parties are disconnected, and it is not possible for the controlling party to disconnect only one participant (that participant must choose to hang up).

I.12.4. The `callId` attribute

When getting the *callId* attribute, the user agent MUST return the call id.

I.12.5. The `serviceId` attribute

When getting the *serviceId* attribute, the user agent MUST return the telephony service id of the telephony service associated with this call.

I.12.6. The `state` attribute

When getting the *state* attribute, the user agent MUST return the CallState value that represents the state of for the Call.

I.12.7. Event handlers

The following are the event handlers are exposed by the CallHandler interface. The event type is `Event`.

event handler	event name
onstatechange	*statechange*
onerror	*error*

I.13. TELEPHONYCALL INTERFACE

Defines the object structure for controlling calls.

readonly attribute DOMString? remoteParty
readonly attribute DOMString? conferenceId
void accept()
void redirect(DOMString remoteParty)
void transfer(DOMString thirdParty)
ConferenceCall createConference()
attribute EventHandler ondialing
attribute EventHandler onalerting
attribute EventHandler onaccepted
attribute EventHandler onconnecting
attribute EventHandler onactive
attribute EventHandler ondisconnecting
attribute EventHandler ondisconnected
attribute EventHandler onholding
attribute EventHandler onheld
attribute EventHandler onresuming
attribute EventHandler onredirecting
attribute EventHandler ontransferring
attribute EventHandler onjoining
attribute EventHandler onmultiparty
attribute EventHandler onsplitting

I.13.1. The `remoteParty` attribute

When getting the *remoteParty* attribute, the user agent MUST return the remote party id (e.g. telephone number) of the call participant. If not available (e.g. callerId has been hidden), return `null`.

I.13.2. The `conferenceId` attribute

When getting the *conferenceId* attribute, if this call is managed as a part of a multiparty call then the user agent MUST return the value of the `conferenceId` attribute of the ConferenceCall multiparty call to which this call is part of. Otherwise, return `null`.

I.13.3. The `accept()` method

The *accept()* method accepts an incoming or waiting telephony call. When invoked, the user agent MUST run the following steps:

1. If `state` is not equal to "incoming" or "waiting", then throw an `InvalidStateError` exception.
2. Otherwise make a request to the telephony system to accept the call. If the request is acknowledged then set `state` to "accepted".

I.13.4. The `redirect()` method

The telephony service in use needs to have the call deflection feature enabled in order for this method to succeed. For instance, in GSM, the Call Deflection supplementary service needs to be active.

The *redirect()* method initiates deflecting an incoming or waiting telephone call to a remote party. The method takes one argument, which represents the remote party to which the call is redirected. When invoked, the user agent MUST run the following steps:

1. If the state is not "incoming" or "waiting", then throw an `InvalidStateError` exception.
2. Otherwise make a request to the telephony system to redirect the call to the number indicated in the *remoteParty* parameter, return, and continue these steps asynchronously.
3. If the request is acknowledged, then set `state` to "redirecting".
4. When the telephony service is notified that the call has been successfully redirected it MUST set `state` to "disconnected".

I.13.5. The `transfer()` method

The telephony service needs to have the call transfer feature enabled in order for this method to succeed. For instance, in GSM, the Call Transfer supplementary service needs to be active.

The *transfer()* method Initiates transferring the call to a new call between the remote party of this call and another remote party, then disconnects the call. The method takes one argument, which represents the remote party to which the call is transferred. When invoked, the user agent MUST run the following steps:

1. If `state` is not equal to "active" or "held", then throw an `InvalidStateError` exception.
2. Otherwise make a request to the telephony system to transfer the call to the number indicated in the *thirdParty* parameter. Note that in GSM this requires putting the current call on hold, dialing a new call to the third party, then initiating the call transfer procedure. If there is an error, the original call MUST be resumed and then queue a task to fire a simple

event named "error" without modifying the call state. If the request is acknowledged, then set `state` to "transferring".
3. When the telephony service is notified that the call has been successfully transferred and the original call is disconnected, it MUST set `state` to "disconnected".

I.13.6. The `createConference()` method

The *createConference()* method Creates a multiparty call from the current call. On cellular telephony services, this happens by initiating merging the active and held calls into a multiparty call. Using this method MUST be the only way to create a ConferenceCall object. When invoked, the user agent MUST run the following steps:

1. Let *telCall* be the telephony call on which this method was invoked.
2. Set the `state` of *telCall* to "joining".
3. Then, in cellular telephony services, make a request to the telephony system to join the active and held calls into a multiparty call.
4. If one of the calls is already a ConferenceCall object, use it as *confCall* in these steps.
5. Otherwise create a new ConferenceCall object with a unique conference id, named *confCall*.
6. Set the `state` of *confCall* to "joining".
7. Return the *confCall* object, and continue these steps asynchronously.
8. If the request to the telephony system is successful,
    1. set the `conferenceId` attribute of the participating calls to the unique identifier generated for the multiparty call *confCall*.
    2. Add the calls participating in the multiparty call to the calls attribute of the *confCall* object, remove them from the calls array of the TelephonyManager object, and set their `state` to `'multiparty'`.
    3. Add the *confCall* object to the calls array of the TelephonyManager object.
    4. Set the `state` of *confCall* to "active".
    5. Queue a task to fire an event named `participantadded` at *confCall*. object
    6. Follow the state changes of *confCall* through the state change events.

I.13.7. Event handlers

To handle changes in telephony state, the TelephonyCall interface implements the following event handlers. The event type for the events is `Event`.

event handler	event name / telephony state
`ondialing`	dialing
`onalerting`	alerting

onaccepted	accepted
onconnecting	connecting
onactive	active
ondisconnecting	disconnecting
ondisconnected	disconnected
onholding	holding
onheld	held
onresuming	resuming
onredirecting	redirecting
ontransferring	transferring
onjoining	joining
onmultiparty	multiparty
onsplitting	splitting

I.14. CONFERENCECALL INTERFACE

Describes the object controlling multiparty calls, and managing the TelephonyCall objects participating in the multiparty call.

readonly attribute DOMString conferenceId
readonly attribute TelephonyCall[] calls
void split(TelephonyCall participantCall)
attribute EventHandler onparticipantadded
attribute EventHandler onparticipantremoved
attribute EventHandler onjoining
attribute EventHandler onactive
attribute EventHandler onsplitting
attribute EventHandler onholding
attribute EventHandler onheld
attribute EventHandler onresuming
attribute EventHandler ondisconnecting
attribute EventHandler ondisconnected

I.14.1. The conferenceId attribute

When getting the *conferenceId* attribute, the user agent MUST return the conference identifier unique in the system and in call history. It MUST NOT be the empty string.

I.14.2. The `calls` attribute

When getting the *calls*, the user agent MUST return the array of TelephonyCall objects managed by the multiparty call object.

I.14.3. The code>split() method

The *split()* method requests the telephony system split the specified participant TelephonyCall object, activate it and put this multiparty call on hold. The method takes one argument, which represents the TelephonyCall object of the call participant to be split from the multiparty call. When invoked, the user agent MUST run the following steps:

1. If the provided *participantCall* does not identify a valid TelephonyCall object which is part of this multiparty call, then an throw an `InvalidModificationError` exception.
2. Otherwise, set the `state` of the *participantCall* and that of this ConferenceCall object to "splitting".
3. Make a request to the telephony system to split the call participant from the multiparty call.
4. Return, and continue these steps asynchronously.
5. If the request was successful, the telephony system will put the multiparty call on hold and activate the split call. The implementation MUST follow the state transitions on the calls as described in this specification.
6. Reset the `conferenceId` of the split call to `null`.

In CDMA, only 3-way calling is supported. This is an IS-41 CN limitation (see Network Interworking between GSM-MAP and TIA-41, 3GPP2 document, p. 1-50 provides a comparison of multiparty call and 3-way call.

I.14.4. Event handlers

To handle changes in telephony state, the ConferenceCall interface implements the following event handlers. The event type for the events is `Event`.

event handler	event name/telephony state
`onparticipantadded`	participantadded
`onparticipantremoved`	participantremoved
`onjoining`	joining
`onactive`	active
`onsplitting`	splitting
`onholding`	holding
`onheld`	held

onresuming	resuming
ondisconnecting	disconnecting
ondisconnected	disconnected

I.15. SECURITY AND PRIVACY CONSIDERATIONS

To be improved. See bug 26.

This API provides access to a potentially dangerous and valuable feature of a device. As a result, misuse of the API would have a large cost to users and other system stakeholders. This API should, therefore, not be implemented without careful consideration of security and privacy issues.

This section provides a limited overview of security and privacy considerations relevant for this API. It includes a set of threats to users and other stakeholders, as well as requirements for mitigating them.

However, this section cannot cover all of the potential threats, nor can it reflect the context in which a conformant implementation may be operating. As a result, this security section should be considered only the starting point for implementers.

I.15.1. Threats

The following list of threats should be considered by the implementer. Note that these are not given in any order.

- The API could be used by a malicious application to deny other system applications access to the device's telephony services, creating an availability problem. This is a safety, as well as security, concern.
- The API could be used by a malicious application as part of a distributed denial of service attack, making frequent calls to a remote call system such as an emergency response number.
- The API could be used by an application to list the telephone numbers that the end user has called and is, at any time, calling. This information ought to be considered private, and could also be used as part of a social engineering attack, or for identity theft.
- The API could be used to make unwanted calls to premium-rate telephone numbers. A malicious application could use this to earn money at the user's expense. Similarly, this API could be misused to enrol the user into a premium-rate calling service, which would then charge the end user when calls are received.
- The API could be used to make unwanted advertising calls, in a similar manner to spam email campaigns. When combined with access to the user's contact list, this would be both expensive and embarassing for the user, and could result in their telephony service being terminated by the network operator.

- The API could be used by a malicious application to make telephone calls impersonating the end user, or as part of a process to defeat a two-factor authentication system.
- A poorly implemented application could misuse this API to make unnecessary or unexpected calls, costing the user money or embarassing them.
- This API could be used to call a number other than the one that the user was expecting, routing calls to an unknown man-in-the-middle. This could be used to eavesdrop on the user. When used in combination with recordings from the microphone, this API could be used to covertly survey the end user.
- This API could be used to send USSD messages to the service provider and invoke functions such as wiping the handset or accessing security settings.
- The API could be misused to access the user's voicemail recordings.
- The API could be misused as part of a DDoS attack on an operator or service provider, flooding the network with calls at certain times.
- The API could cost the end user money by making outgoing calls when the user is roaming, or on an expensive network.

I.15.2. Mitigations

The following mechanisms may be employed to help an implementer mitigate the threats outlined in the previous section.

- The user agent should only expose this API to *privileged* applications, as defined in the Runtime and Security Model.
- The user agent should only expose this API to applications which were distributed by an institution that the handset recognises as a valid source. For example, the API might only be accessible to applications distributed by the handset manufacturer.
- All applications with access to this API should be reviewed before they are made available. A mechanism for remote update of applications with access to this API should be provided to allow for identified security issues to be fixed.
- The user agent should maintain the integrity of any application with access to this API when initially downloaded, as well as when it is stored offline.
- The user agent should only expose this API to downloaded, offline applications which are not modifiable by external web servers. A restrictive content security policy should be used to enforce that application with external content (such as scripts) cannot access this API.
- The API implementation should have different behaviour when used with premium-rate numbers. Accessing premium-rate numbers may require an additional permission to be listed in the manifest, a different (or additional) warning to be displayed to users, or place an additional requirement on the valid distributors of the application. It is up to the

implementing user agent to identify whether a remote party identifier is premium-rate or not.
- The API implementation should have different behaviour when the user is roaming on a network with a different (less favourable) service-level agreement. For example, presenting a different warning to the user, or denying access to this API from certain applications altogether.
- User consent must be captured when a call is made. For example, the user must press a 'dial' button, or equivalent, before the call is placed. The user must also be shown the recipients of the call, and the numbers that have and will be dialled as part of placing it.
- It should be obvious, visually, when a call is being invoked, is in progress, and has ended. This should be visible to the end user and it must not be possible for applications to hide or obscure this indicator.
- The user agent should introduce rate limiting to prevent an application from making too many calls in too short a period of time.

I.15.3. User interaction guidelines

TODO

I.16. MANAGING CALL HISTORY

The CallHistoryEntry interface describes the minimum set of properties which a user agent would need to support for call history entries. For multiparty call there needs to be a separate CallHistoryEntry object for each call participant, sharing the same value for the conferenceId attribute.

It is up the the implementations and applications how to store and access call history. This document only specifies the minimum content of the data to be saved.

I.16.1. `CallHistoryEntry` interface

readonly attribute DOMString remoteParty
readonly attribute DOMString serviceId
readonly attribute DOMString? conferenceId
readonly attribute Date startTime
readonly attribute unsigned long long duration
readonly attribute CallDirection direction
readonly attribute DisconnectReason? disconnectReason
readonly attribute boolean emergency

I.16.2. The `remoteParty` attribute

When getting the *remoteParty* attribute, the user agent MUST return the remote party id (e.g. telephone number) of the call participant.

I.16.3. The `serviceId` attribute

When getting the *serviceId* attribute, the user agent MUST return the telephony service id of the telephony service used for the call.

I.16.4. The `conferenceId` attribute

When getting the *conferenceId* attribute, the user agent MUST return the conference id of the call, if the call has participated in a multiparty call. Otherwise, return `null`. string.

I.16.5. The `startTime` attribute

When getting the *startTime* attribute, the user agent MUST return the starting time of the call, measured from when the call is in active state.

I.16.6. The `duration` attribute

When getting the *duration* attribute, the user agent MUST return the duration of the call expressed in milliseconds.

I.16.7. The `direction` attribute

When getting the *direction* attribute, the user agent MUST return the CallDirection.

I.16.8. The `disconnectReason` attribute

When getting the *disconnectReason* attribute, the user agent MUST return the DisconnectReason if available, or return `null` otherwise.

I.16.9. The `emergency` attribute

When getting the *emergency* attribute, the user agent MUST return `true` if the call was an emergency call, or `false` otherwise.

I.17. CALLDIRECTION ENUM

dialed
The call has been dialed.

received
The call has been received.

missed
The call has been missed.

missed-new
The call was a missed call not seen yet by the user.

I.18. CHANGES

The following is a list of substantial changes to the document. For a complete list of changes, see the change log on Github. You can also view the recently closed bugs.

- No changes yet.

I.19. ACKNOWLEDGEMENTS

The editors would like to express their gratitude to the Mozilla B2G Team for their technical guidance, implementation work and support, especially to Ben Turner and Jonas Sicking, the authors of the *B2G WebTelephony API*. Also, thanks to Denis Kenzior (*ofono* maintainer) and Oleg Zhurakivskyy of Intel Open Source Technology Center, and many others for their advice and support.

J. TCP AND UDP SOCKET API

W3C Editor's Draft 13 October 2014

This version:
http://www.w3.org/2012/sysapps/telephony/

Latest published version:
http://www.w3.org/TR/telephony/

Latest editor's draft:
http://www.w3.org/2012/sysapps/telephony/

Editors:
Claes Nilsson, Sony Mobile

This API provides interfaces to raw UDP sockets, TCP Client sockets and TCP Server sockets.

This specification is based the Streams API, [[!STREAMS]]. Note that the Streams API is work in progress and any changes made to Streams may impact the TCP and UDP Socket API specification. However, it is the editor's ambition to continously update the TCP and UDP API specification to be aligned with the latest version the Streams API.

This is a note on error handling.

When using promises rejection reasons should always be instances of the ECMAScript Error type such as DOMException or the built in ECMAScript error types. See Promise rejection reasons. In the TCP and UDP Socket API the error names defined in WebIDL Exceptions are used. If additional error names are needed an update to Github WebIDL should be requested though a Pull Request.

This is a note on data types of TCP and UDP to send and receive.

In the previous version of this API the send() method accepted the following data types for the data to send: DOMString,Blob, ArrayBuffer or ArrayBufferView. This was aligned with the send() method for Web Sockets. In this Streams API based version only ArrayBuffer is accepted as type for data to send. The reason is that encoding issues in a Streams based API should instead be handled by a transform stream.

J.1. INTRODUCTION

Use this API to send and receive data over the network using TCP or UDP.

Examples of use cases for this API are:

- An email client which communicates with SMTP, POP3 and IMAP servers
- An irc client which communicates with irc servers
- Implementing an ssh app
- Communicating with existing consumer hardware, like internet connected TVs
- Game servers
- Peer-to-peer applications
- Local network multicast service discovery, e.g. UPnP/SSDP and mDNS

This specification defines conformance criteria that apply to a single product: the *user agent* that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [[!WEBIDL]], as this specification uses that specification and terminology.

J.2. TERMINOLOGY

The *`Promise`* interface provides asynchronous access to the result of an operation that is ongoing, has yet to start, or has completed, as defined in [[!ES6]].

J.3. SECURITY AND PRIVACY CONSIDERATIONS

This API must be only exposed to trusted content.

There is ongoing work on trust and permissions in W3C. For example see Workshop on trust and permissions for Web applications 3–4 September 2014, Paris, France. The assumption is that this API must only be exposed to trusted content according to a security model based on existing web security mechanisms such as tls/ssl, signed manifests, csp, etc. The details of that security model as such is out of scope for this specification as this model should apply to any security and privacy sensitive API.

J.4. INTERFACE UDPSOCKET

The UDPSocket interface defines attributes and methods for UDP communication

```
//
// This example shows a simple implementation of UPnP-SSDP M-SEARCH
// discovery using a multicast UDPSocket
//

var address = '239.255.255.250',
    port = '1900',
    serviceType = 'upnp:rootdevice',
    rn = '\r\n',
    search = '';

//  Create a new UDP client socket
var mySocket = new UDPSocket();

// Build an SSDP M-SEARCH multicast message
search += 'M-SEARCH * HTTP/1.1' + rn;
search += 'ST: ' + serviceType + rn;
search += 'MAN: "ssdp:discover"' + rn;
```

```
        search += 'HOST: ' + address + ':' + port + rn;
        search += 'MX: 10';


        // Receive and log SSDP M-SEARCH response messages
        function receiveMSearchResponses() {

          // While data in buffer, read and log UDP message
          while (mySocket.readable.state === "readable") {
            var msg = mySocket.readable.read();
            console.log ('Remote address: ' + msg.remoteAddress +
                         ' Remote port: ' + msg.remotePort +
                         'Message: ' + ab2str(msg.data));
              // ArrayBuffer to string conversion could also be done by piping
              // through a transform stream. To be updated when the Streams API
              // specification has been stabilized on this point.
          }

          // Wait for SSDP M-SEARCH responses to arrive
          mySocket.readable.wait().then(
            receiveMSearchResponses,
            e => console.error("Receiving error: ", e);
          );
        }

        // Join SSDP multicast group
        mySocket.joinMulticast(address);

        // Send SSDP M-SEARCH multicast message
        mySocket.writeable.write(
          {data : str2ab(search),
           remoteAddress : address,
           remotePort : port
          }).then(
            () => {
              // Data sent sucessfully, wait for response
              console.log('M-SEARCH Sent');
              receiveMSearchResponses();
            },
            e => console.error("Sending error: ", e);
        );

        // Log result of UDP socket setup.
        mySocket.opened.then(
          () => {
            console.log("UDP socket created sucessfully");
          },
          e =>console.error("UDP socket setup failed due to error: ", e);
        );

        // Handle UDP socket closed, either as a result of the application
        // calling mySocket.close() or an error causing the socket to be
           closed.
        mySocket.closed.then(
          () => {
             console.log("Socket has been cleanly closed");
          },
          e => console.error("Socket closed due to error: ", e);
        );
```

readonly attribute DOMString? localAddress

The IPv4/6 address of the local interface, e.g. wifi or 3G, that the UDPSocket object is bound to. Can be set by the constructor's `options` argument's `localAddress` member. If this member is not present but the `remoteAddress` member is present, the user agent binds the socket to a local IPv4/6 address based on the routing table and possiby a preselect default local interface to use for the selected `remoteAddress`. Else, i.e. neither the `localAddress` or the `remoteAddress` members are present in the constructor's `options` argument, the `localAddress` attribute is set to `null`.

readonly attribute unsigned short? localPort

The local port that the UDPSocket object is bound to. Can be set by the `options` argument in the constructor. If not set the user agent binds the socket to an ephemeral local port decided by the system and this attribute is null.

readonly attribute DOMString? remoteAddress

The default remote host name or IPv4/6 address that is used for subsequent send() calls. Null if not stated by the options argument of the constructor.

readonly attribute unsigned short? remotePort

The default remote port that is used for subsequent send() calls. Null if not stated by the options argument of the constructor

readonly attribute boolean addressReuse

`true` allows the socket to be bound to a local address/port pair that already is in use. Can be set by the `options` argument in the constructor. Default is `true`.

readonly attribute boolean loopback

Only applicable for multicast.`true` means that sent multicast data is looped back to the sender. Can be set by the `options` argument in the constructor. Default is `false`.

readonly attribute SocketReadyState readyState

The state of the UDP Socket object. A UDP Socket object can be in "open" "opening" or "closed" states. See enum SocketReadyState for details.

readonly attribute Promise opened

Detects the result of the UDP socket creation attempt.Returns the `openedPromise` that was created in the `UDPSocket` constructor.

readonly attribute Promise closed

Detects when the UDP socket has been closed, either cleanly by the client application calling `close()`) or through an error situation, e.g. network contact lost. Returns the `closedPromise` that was created in the `UDPSocket` constructor.

readonly attribute ReadableStream readable
The object that represents the UDP socket's source of data, from which you can read. [[!STREAMS]]

readonly attribute WriteableStream writeable
The object that represents the UDP socket's destination for data, into which you can write. [[!STREAMS]]

Promise close()

Closes the UDP socket. Returns the `closedPromise` that was created in the `UDPSocket` constructor.

void joinMulticast()

Joins a multicast group identified by the given address.

Note that even if the socket is only sending to a multicast address, it is a good practice to explicitely join the multicast group (otherwise some routers may not relay packets).

DOMString multicastGroupAddress
The multicast group address.

void leaveMulticast()

Leaves a multicast group membership identified by the given address.

DOMString multicastGroupAddress
The multicast group address.

When the UDPSocket constructor is invoked, the User Agent MUST run the following steps:

1. Create a new UDPSocket object ("`mySocket`").
2. If the `options` argument's `remoteAddress` member is present and it is a valid host name or IPv4/6 address then set the `mySocket.remoteAddress` attribute (default remote address) to the requested address. Else, if the `remoteAddress` member is present but it is not a valid host name or IPv4/6 address then throw DOMException `InvalidAccessError` and abort the remaining steps. Otherwise, if the `options` argument's `remoteAddress` member is absent then set the `mySocket.remoteAddress` attribute (default remote address) to `null`.
3. If the `options` argument's `remotePort` member is present and it is a valid port number then set the `mySocket.remotePort` attribute (default remote port) to the requested port. Else, if the `remotePort` member is present but it is not a valid port number then throw DOMException

`InvalidAccessError` and abort the remaining steps. Otherwise, if the `options` argument's `remotePort` member is absent then set the `mySocket.remotePort` attribute (default port number) to `null`.

4. If the `options` argument's `localAddress` member is present and the `options` argument's `remoteAddress` member is present, execute the following step:
    - If the `options` argument's `localAddress` member is a valid IPv4/6 address for a local interface that can be used to connect to the selected `remoteAddress` (according to the routing table) bind the socket to this local IPv4/6 address and set the `mySocket.localAddress` attribute to this addres. Else, if the `localAddress` member is present but it is not a valid local IPv4/6 address for a local interface that can be used to connect to the selected `remoteAddress`, throw DOMException `InvalidAccessError` and abort the remaining steps.

    Else, if the `options` argument's `localAddress` member is present and the `options` argument's `remoteAddress` member is absent, execute the following step:
    - If the `options` argument's `localAddress` member is a valid IPv4/6 address for a local interface on the device bind the socket to this local IPv4/6 address and set the `mySocket.localAddress` attribute to this addres. Else, if the `localAddress` member is present but it is not a valid local IPv4/6 address for a local interface on the device, throw DOMException `InvalidAccessError` and abort the remaining steps. Note that binding the UDPSocket to a certain local interface means that the socket can only be used to send UDP datagrams to peers reachable through this local interface.

    Else, if the `options` argument's `localAddress` member is absent, and the `options` argument's `remoteAddress` member is present, execute the following steps:
    1. Use the routing table to determine the local interface(s) that can be used to send datagrams to the selected `remoteAddress`. If no local interface can be used to send datagrams to the selected `remoteAddress`, throw DOMException `InvalidAccessError` and abort the remaining steps.
    2. If the routing table states that more than one local interface can be used to send datagrams to the selected `remoteAddress` bind the socket to the IPv4/6 address of the "default" local interface to use for the selected `remoteAddress`. The selection of a "default" local interface is out of scope for this specification.
    3. Set the `mySocket.localAddress` attribute to the local address that the socket is bound to.

    Else, i.e. the `options` argument's `localAddress` member is absent, and the `options` argument's `remoteAddress` member is absent, execute the following step:
    - Set the `mySocket.localAddress` attribute to `null`.
5. If the `options` argument's `localPort` member is absent then bind the socket to an ephemeral local port decided by the system and set the

`mySocket.localPort` attribute to null. Otherwise, bind the socket to the requested local port and set the `mySocket.localPort` attribute to the local port that the socket is bound to.

6. Set the `mySocket.addressReuse` attribute to the value of the `options` argument's `addressReuse` member if it is present or to `true` if the `options` argument's `addressReuse` member is not present.
7. If the `options` argument's `loopback` member is present then set the `mySocket.loopback` attribute to the value of this field. Else set this attribute to `false`.
8. Set the `mySocket.readyState` attribute to "opening".
9. Create a new promise, "`openedPromise`", and store it so it can later be returned by the `opened` property.
10. Create a new promise, "`closedPromise`", and store it so it can later be returned by the `closed` property and the `close` method.
11. Let the `mySocket.readable` attribute be a new ReadableStream object, [[!STREAMS]]. The User Agent MUST implement the adaptation layer to [[!STREAMS]] for this new ReadableStream object through implementation of a number of functions that are given as input arguments to the constructor and called by the [[!STREAMS]] implementation. The semantics for these functions are described below:
    - The constructor's `start()` function is called immediately by the [[!STREAMS]] implementation. The `start()` function MUST run the following steps:
        1. Setup the UDP socket to the bound local and remote address/port pairs in the background (without blocking scripts) and return `openedPromise`.
        2. When the UDP socket has been successfully setup the following steps MUST run:
            1. Change the `mySocket.readyState` attribute's value to "open".
            2. Resolve `openedPromise` with `undefined`.

      The following internal methods of the ReadableStream are arguments of the constructor's `start()` function and MUST be called by the `start()` function implementation according to the following steps:
        - The `enqueue()` argument of `start()` is a function that pushes received data into the internal buffer.

          When a new UDP datagram has been received the following steps MUST run:
            1. Create a new instance of UDPMessage.
            2. Set the UDPMessage object's `data` member to a new read-only `ArrayBuffer` object whose contents are the received UDP datagram [[!TYPED-ARRAYS]].
            3. Set the `remoteAddress` member of the UDPMessage object to the source address of the received UDP datagram.
            4. Set the `remotePort` member of the UDPMessage object to the source port of the received UDP datagram.

5. Call `enqueue()` to push the UDPMessage object into the internal [[!STREAMS]] receive buffer. Note that `enqueue()` returns false if the high watermark of the buffer is reached. However, as there is no flow control mechanism in UDP the flow of datagrams can't be stopped. The `enqueue()` return value should therefore be ignored. This means that datagrams will be lost if the internal receive buffer has been filled to it's memory limit but this is the nature of an unreliable protocol as UDP.

- The `error()` argument of `start()` is a function that handles readable stream errors and closes the readble stream.
  Upon detection that the attempt to setup a new UDP socket (`mySocket.readyState` is "opening") has failed, e.g. because the local address/port pair is already in use and `mySocket.addressReuse` is `false`, the following steps MUST run:
  1. Call `error()` with DOMException "`NetworkError`".
  2. Reject `openedPromise` with DOMException "`NetworkError`".
  3. Reject `closedPromise` with DOMException "`NetworkError`".
  4. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

  Upon detection that there is an error with the established UDP socket (`mySocket.readyState` is "open"), e.g. network connection is lost, the following steps MUST run:
  1. Call `error()` with DOMException "`NetworkError`".
  2. Reject `closedPromise` with DOMException "`NetworkError`".
  3. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

  When a new UDP datagram has been received and upon detction that it is not possible to convert the received UDP data to `ArrayBuffer`, [[!TYPED-ARRAYS]], the following steps MUST run:
  1. Call `error()` with `TypeError`.
  2. Reject `closedPromise` with `TypeError`.
  3. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

- The constructor's `pull()` function MUST be omitted as there is no flow control mechanism in UDP and the flow of datagrams cannot be stopped and started again.
- The constructor's `cancel()` function input argument is called by the [[!STREAMS]] implementation when the ReadbleStream

should be canceled. For UDP this means that the UDP socket should be closed for reading and writing. The `cancel()` function MUST run the following steps:

1. If `mySocket.readyState` is "closed" then do nothing and abort the remaning steps.
2. If `mySocket.readyState` is "opening" then fail the UDP socket setup process, reject `openedPromise` with DOMException `AbortError` and set the `mySocket.readyState` attribute to "closed".
3. If `mySocket.readyState` is "open" the the following steps MUST run:
    1. Call `mySocket.writeable.close()` to assure that any buffered send data is sent.
    2. Set the `mySocket.readyState` attribute's value to "closed".
    3. Resolve `closedPromise` with `undefined` and release any underlying resources associated with this socket.

If the constructor's `strategy` argument is omitted the Default strategy for Readable Streams applies. Currently this means that the ReadableStream object goes to "readable" state after 1 chunk has been enqueued to the internal ReadableStream object's input buffer. An application should call .wait() to be notified when the state changes to "readable". To be further investigated which ReadableStreamStrategy that should be applied to UDP.

12. Let the `mySocket.writeable` attribute be a new WritableStream object, [[!STREAMS]]. The User Agent MUST implement the adaptation layer to [[!STREAMS]] for this new WritableStream object through implementation of a number of functions that are given as input arguments to the constructor and called by the [[!STREAMS]] implementation. The semantics for these functions are described below:
    - The constructor's `start()` function MUST run the following steps:
        1. Create a new promise, "`writableStartPromise`".
        2. If the attempt to create a new UDP socket (see the description of the semantics for the `mySocket.readable` attribute constructor's `start()` function) succeded resolve `writableStartPromise` with `undefined`, else reject `writableStartPromise` with DOMException `"NetworkError"`.
    - The constructor's `write(chunk)` function is called by the [[!STREAMS]] implementation to write UDP data. The `write()` function MUST run the following steps:
        1. Create a new promise, "`writePromise`"
        2. Let the `chunk` argument be the result of converting data to an UDPMessage (per [[!WEBIDL]] dictionary conversion).

3. If no default remote address was specified in the UDPSocket's constructor `options` argument's `remoteAddress` member and the UDPMesssage object's `remoteAddress` member is not present or null then throw DOMException `InvalidAccessError` and abort these steps.
4. If no default remote port was specified in the UDPSocket's constructor `options` argument's `remotePort` member and the UDPMesssage object's `remotePort` member is not present or null then throw DOMException `InvalidAccessError` and abort these steps.
5. Send UDP data with data passed in the `data` member of the UDPMessage object. The destination address is the address defined by the UDPMesssage object's `remoteAddress` member if present, else the destination address is defined by the UDPSocket's constructor `options` argument's `remoteAddress` member. The destination port is the port defined by the UDPMesssage object's `remotePort` member if present, else the destination port is defined by the UDPSocket's constructor `options` argument's `remotePort` member.
6. If sending succeed resolve `writePromise` with `undefined`, else reject `writePromise` with DOMException `"NetworkError"`.

◦ The constructor's `close()` and `abort()` functions MUST be omitted as it is not possible to just close the writable side of a UDP socket.

  If the constructor's `strategy` argument is omitted the Default strategy for Writable Streams applies. Currently this means that the WriteableStream object goes to "waiting" state after 1 chunk has been written to the internal WriteableStream object's output buffer. This means that the application should call .wait() to be notified of when the state changes to "writable", i.e. the queued chunk has been written to the remote peer and more data chunks could be written. To be further investigated which WritableStreamStrategy that should be applied to UDP.

13. Return the newly created `UDPSocket` object (`"mySocket"`) to the application.

The *`close`* method when invoked MUST run the following steps:

1. Call mysocket.readable.cancel(reason). (Reason codes TBD.)
2. Return `closedPromise`.

J.5. INTERFACE TCPSOCKET

The TCPSocket interface defines attributes and methods for TCP communication

```
//
// This example shows a simple TCP echo client.
// The client will send "Hello World" to the server on port 6789 and log
// what has been received from the server.
//

//  Create a new TCP client socket and connect to remote host
var mySocket = new TCPSocket("127.0.0.1", 6789);

// Send data to server
mySocket.writeable.write("Hello World").then(
    () => {

        // Data sent sucessfully, wait for response
        console.log("Data has been sent to server");
        mySocket.readable.wait().then(
            () => {

                // Data in buffer, read it
                console.log("Data received from server:"
                             + mySocket.readable.read());

                // Close the TCP connection
                mySocket.close();
            },

            e => console.error("Receiving error: ", e);
        );
    },
    e => console.error("Sending error: ", e);
);

// Signal that we won't be writing any more and can
// close the write half of the connection.
mySocket.halfClose();

// Log result of TCP connection attempt.
mySocket.opened.then(
  () => {
    console.log("TCP connection established sucessfully");
  },
  e =>console.error("TCP connection setup failed due to error: ", e);
);

// Handle TCP connection closed, either as a result of the application
// calling mySocket.close() or the other side closed the TCP
// connection or an error causing the TCP connection to be closed.
mySocket.closed.then(
  () => {
     console.log("TCP socket has been cleanly closed");
  },
  e => console.error("TCP socket closed due to error: ", e);
);
```

readonly attribute DOMString remoteAddress

The host name or IPv4/6 address of the peer as stated by the remoteAddress argument in the constructor.

readonly attribute unsigned short remotePort

The port of the peer as stated by the remotePort argument in the constructor.

readonly attribute DOMString localAddress

The IPv4/6 address of the local interface, e.g. wifi or 3G, that the TCPSocket object is bound to. Can be set by the `options` argument in the constructor. If not set the user agent binds the socket to an IPv4/6 address based on the routing table and possibly a preselect default local interface to use for the selected `remoteAddress`.

readonly attribute unsigned short localPort

The local port that the TCPSocket object is bound to. Can be set by the `options` argument in the constructor. If not set the user agent binds the socket to an ephemeral local port decided by the system.

readonly attribute boolean addressReuse

`true` allows the socket to be bound to a local address/port pair that already is in use. Can be set by the `options` argument in the constructor. Default is `true`.

readonly attribute boolean noDelay

`true` if the Nagle algorithm for send coalescing, [[!NAGLE]], is disabled. Can be set by the `options` argument in the constructor. Default is `true`.

readonly attribute SocketReadyState readyState

The state of the TCP Socket object. See enum SocketReadyState for details.

readonly attribute Promise opened

Detects the result of the TCP connection attempt with the remote peer. Returns the `openedPromise` that was created in the `TCPSocket` constructor.

readonly attribute Promise closed

Detects when the TCP connection has been closed, either cleanly (initiated either by the server, or by the client application calling `close()`) or through an error situation. Returns the `closedPromise` that was created in the `TCPSocket` constructor.

readonly attribute ReadableStream readable

The object that represents the TCP socket's source of data, from which you can read. [[!STREAMS]]

readonly attribute WriteableStream writeable

The object that represents the TCP socket's destination for data, into which you can write. [[!STREAMS]]

Promise close()

Closes the TCP socket. Returns the `closedPromise` that was created in the `TCPSocket` constructor.

void halfClose()

Half closes the TCP socket.

When the TCPSocket constructor is invoked, the User Agent MUST run the following steps:

1. Create a new TCPSocket object ("`mySocket`").
2. If the `remoteAddress` argument is not a valid host name or IPv4/6 address and/or the `remotePort` argument is not a valid port number then throw DOMException `"InvalidAccessError"` and abort the remaining steps, else set the `mySocket.remoteAddress` and `mySocket.remotePort` attributes to the requested values.
3. If the `options` argument's `localAddress` member is present and it is a valid IPv4/6 address for a local interface that can be used to connect to the selected `remoteAddress` (according to the routing table) bind the socket to this local IPv4/6 address and set the `mySocket.localAddress` attribute to this addres. Else, if the `localAddress` member is present but it is not a valid local IPv4/6 address for a local interface that can be used to connect to the selected `remoteAddress` then throw DOMException `"InvalidAccessError"` and abort the remaining steps.
   Otherwise, if the `options` argument's `localAddress` member is absent, execute the following steps:
   1. Use the routing table to determine the local interface(s) that can be used to connect to the selected `remoteAddress`. If no local interface can be used to connect to the selected `remoteAddress` then throw DOMException `"InvalidAccessError"` and abort the remaining steps.
   2. If the routing table states that more than one local interface can be used to connect to the selected `remoteAddress` bind the socket to the IPv4/6 address of the "default" local interface to use for the selected `remoteAddress`. The selection of a "default" local interface is out of scope for this specification.
   3. Set the `mySocket.localAddress` attribute to the local address that the socket is bound to.
4. If the `options` argument's `localPort` member is absent then bind the socket to an ephemeral local port decided by the system and set the `mySocket.localPort` attribute to this port. Otherwise, bind the socket to the requested local port and set the `mySocket.localPort` attribute to the local port that the socket is bound to.

5. Set the `mySocket.addressReuse` attribute to the value of the `options` argument's `addressReuse` member if it is present or to `true` if the `options` argument's `addressReuse` member is not present.
6. Set the `mySocket.noDelay` attribute to the value of the `options` argument's `noDelay` member if it is present or to `true` if the `options` argument's `noDelay` member is not present.
7. Set the `mySocket.readyState` attribute to "opening".
8. Create a new promise, "`openedPromise`", and store it so it can later be returned by the `opened` property.
9. Create a new promise, "`closedPromise`", and store it so it can later be returned by the `closed` property and the `close` method.
10. Let the `mySocket.readable` attribute be a new ReadableStream object, [[!STREAMS]]. The User Agent MUST implement the adaptation layer to [[!STREAMS]] for this new ReadableStream object through implementation of a number of functions that are given as input arguments to the constructor and called by the [[!STREAMS]] implementation. The semantics for these functions are described below:
    - The constructor's `start()` function is called immediately by the [[!STREAMS]] implementation. The `start()` function MUST run the following steps:
        1. A TCP connection setup handshake to the requested address and port MUST be performed in the background (without blocking scripts). Return `openedPromise`.
        2. When the TCP connection has been successfully established the following steps MUST run:
            1. Change the `mySocket.readyState` attribute's value to "open".
            2. Resolve `openedPromise` with `undefined`.

      The following functions are arguments of the constructor's `start()` function and MUST be called by the `start()` function implementation according to the following steps:
        - The `enqueue()` argument of `start()` is a function that pushes received data into the internal buffer.
          When new TCP data is received the following steps MUST run:
            1. Create a new read-only `ArrayBuffer` object whose contents are the received TCP data.
            2. Call `enqueue()` to push the `ArrayBuffer` into the internal [[!STREAMS]] receive buffer.
            3. If the high watermark of the buffer is reached `enqueue()` returns false and the TCP flow control MUST be used to stop the data transmission from the remote peer.
        - The `close()` argument of `start()` is a function that closes the readable stream.
          Upon detection that the TCP connection has been closed cleanly (initiated either by the server, or by the client application calling `close()`) through a successful TCP

connection closing handshake the following steps MUST run:

   1. Call `close()`.
   2. Set the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.
   3. Resolve `closedPromise` with `undefined`.

- The `error()` argument of `start()` is a function that handles readable stream errors and closes the readble stream.
  Upon detection that the attempt to create a new TCP socket and establish a new TCP connection (`mySocket.readyState` is "opening") has failed the following steps MUST run:
   1. Call `error()` with DOMException `"NetworkError"`.
   2. Reject `openedPromise` with DOMException `"NetworkError"`.
   3. Reject `closedPromise` with DOMException `"NetworkError"`.
   4. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

  Upon detection that the established TCP connection (`mySocket.readyState` is "open") has been lost the following steps MUST run:
   1. Call `error()` with DOMException `"NetworkError"`.
   2. Reject `closedPromise` with DOMException `"NetworkError"`.
   3. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

  Upon detection that the TCP connection closing handshake failed (`mySocket.readyState` is "closing") has failed the following steps MUST run:
   1. Call `error()` with DOMException `"NetworkError"`.
   2. Reject `closedPromise` with DOMException `"NetworkError"`.
   3. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

  When new TCP data has been received and upon detction that it is not possible to convert the received data to `ArrayBuffer`, [[!TYPED-ARRAYS]], the following steps MUST run:
   1. Call `error()` with `TypeError`.
   2. Reject `closedPromise` with `TypeError`.
   3. Change the `mySocket.readyState` attribute's value to "closed" and release any underlying resources associated with this socket.

- The constructor's `pull()` function is called by the [[!STREAMS]] implementation if the internal buffer has been emptied, but the stream's consumer still wants more data. The `pull()` function MUST run the following steps:
    1. The function MUST resume receiving TCP data through the TCP flow control mechanism.
- The constructor's `cancel()` function input argument is called by the [[!STREAMS]] implementation when the ReadbleStream should be canceled. For TCP this means that the TCP connection should be terminated. The `cancel()` function MUST run the following steps:
    1. If `mySocket.readyState` is "closing" or "closed" then do nothing and abort the remaning steps.
    2. If `mySocket.readyState` is "opening" then fail the connection attempt, reject `openedPromise` with DOMException `AbortError` and set the `mySocket.readyState` attribute to "closing".
    3. If `mySocket.readyState` is "open" then the following steps MUST run:
        1. Call `mySocket.writeable.close()` to assure that any buffered send data is sent before closing the socket.
        2. Set the `mySocket.readyState` attribute's value to "closing".
        3. Initiate TCP closing handshake.

  If the constructor's `strategy` argument is omitted the Default strategy for Readable Streams applies. Currently this means that the ReadableStream object goes to "readable" state after 1 chunk has been enqueued to the internal ReadableStream object's input buffer. An application should call .wait() to be notified when the state changes to "readable". To be further investigated which ReadableStreamStrategy that should be applied to TCP.

11. Let the `mySocket.writeable` attribute be a new WritableStream object, [[!STREAMS]]. The User Agent MUST implement the adaptation layer to [[!STREAMS]] for this new WritableStream object through implementation of a number of functions that are given as input arguments to the constructor and called by the [[!STREAMS]] implementation. The semantics for these functions are described below:
    - The constructor's `start()` function MUST run the following steps:
        1. Create a new promise, "`writableStartPromise`".
        2. If the attempt to create a new TCP socket and establish a new TCP connection (see the description of the semantics for the `mySocket.readable` attribute constructor's `start()` function) succeded resolve `writableStartPromise` with `undefined`, else reject

`writableStartPromise` with DOMException `"NetworkError"`.

- The constructor's `write(chunk)` function is called by the [[!STREAMS]] implementation to write data to the remote peer on the TCP connection. The `write()` function MUST run the following steps:
  1. Create a new promise, "`writePromise`"
  2. Send TCP data with data passed in the `chunk` parameter to the address and port of the recipient as stated by the TCPSocket object constructor's `remoteAddress` and `remotePort` fields. The data in the `chunk` parameter can be of any type.
  3. If sending succeed resolve `writePromise` with `undefined`, else reject `writePromise` with DOMException `"NetworkError"`.
- The constructor's `close()` function is called by the [[!STREAMS]] implementation to close the writable side of the connection, that is a TCP "half close" is performed. The `close()` function MUST run the following steps:
  1. If `mySocket.readyState` is "closing" or "closed" then do nothing.
  2. If `mySocket.readyState` is "opening" then complete the connection attempt. If succesful send FIN and set the `mySocket.readyState` attribute to "halfclosed".
  3. If `mySocket.readyState` is "open" then send FIN and set the `mySocket.readyState` attribute to "halfclosed".

  Note that the Streams implementation will call `close()` after all queued-up writes successfully completed.
- The constructor's `abort()` function is called by the [[!STREAMS]] implementation to abort the writable side of the connection. This function MUST run the same steps as `close()` but note that the Streams implementation will throw away any pending queued up chunks.

  If the constructor's `strategy` argument is omitted the Default strategy for Writable Streams applies. Currently this means that the WriteableStream object goes to "waiting" state after 1 chunk has been written to the internal WriteableStream object's output buffer. This means that the application should call .wait() to be notified of when the state changes to "writable", i.e. the queued chunk has been written to the remote peer and more data chunks could be written. To be further investigated which WritableStreamStrategy that should be applied to TCP.

12. Return the newly created TCPSocket object ("`mySocket`") to the application.

The *`close()`* method when invoked MUST run the following steps:

1. Call `mysocket.readable.cancel(reason)`. (Reason codes TBD.)
2. Return `closedPromise`.

The *`halfClose()`* method when invoked MUST run the following steps:

1. Call `mysocket.writeable.close()`.

J.6. INTERFACE TCPSERVERSOCKET

The TCPServerSocket interface supports TCP server sockets that listens to connection attempts from TCP clients

```
      //
      // This example shows a simple TCP echo server.
      // The server will listen on port 6789 and respond back with whatever
      // has been sent to the server.
      //
       //  Create a new server socket that listens on port 6789
       var myServerSocket = new TCPServerSocket({"localPort": 6789});

       // Listen for connection attempts from TCP clients.
       listenForConnections();
       function listenForConnections() {
         myServerSocket.listen().then(
             connectedSocket => {
             // A connection has been accepted

               console.log ("Connection accepted from address: " +
                            connectedSocket.remoteAddress + " port: " +
                            connectedSocket.remotePort);
               // Wait for data
               waitForData ();
               function waitForData () {
                 connectedSocket.readable.wait().then(
                   () => {
                       // Data in buffer, read it
                       var receivedData = connectedSocket.readable.read();
                       console.log ("Received: " + receivedData);

                       // Send data back
                       connected.writeable.write(receivedData).then(
                         () => {
                           console.log ("Sending data succeeded");
                         },
                         e => console.error("Failed to send: ", e);
                       },
                       // Continue to wait for data
                       waitForData ();
                   },
                   e => {
                     console.error("Error in connection: ", e);
                     // Continue to wait for data
                     waitForData ();
                   }
                 );

               }
```

```
              // Continue to listen for new connections
              listenForConnections();
            },
            e => {
              console.error("A client connection attempt failed: ", e);

              // Continue to listen for new connections
              listenForConnections();
            }

        );
      }

      // Log result of TCP server socket creation attempt.
      myServerSocket.opened.then(
        () => {
          console.log("TCP server socket created sucessfully");
        },
        e =>console.error("TCP server socket creation failed due to error: ", e);
      );

      // Handle TCP server closed, either as a result of the application
      // calling myServerSocket.close() or due to an error.
      myServerSocket.closed.then(
        () => {
           console.log("TCP server socket has been cleanly closed");
        },
        e => console.error("TCP server socket closed due to error: ", e);
      );
```

readonly attribute DOMString localAddress

The IPv4/6 address of the interface, e.g. wifi or 3G, that the TCPServer Socket object is bound to. Can be set by the `options` argument in the constructor. If not set the the server will accept connections directed to any IPv4 address and this atribute is set to `null`.

readonly attribute unsigned short localPort

The local port that the TCPServerSocket object is bound to. Can be set by the `options` argument in the constructor. If not set the user agent binds the socket to an ephemeral local port decided by the system and sets this atribute to `null`.

readonly attribute boolean addressReuse

`true` allows the socket to be bound to a local address/port pair that already is in use. Can be set by the `options` argument in the constructor. Default is `true`.

readonly attribute SocketReadyState readyState

The state of the TCP server object. A TCP server socket object can be in "open", "opening" or "closed" states. See enum SocketReadyState for details.

readonly attribute Promise opened

Detects the result of the TCP server socket opening process when the socket is ready to receive connection attempts from clients. Returns the `openedPromise` that was created in the `TCPServerSocket` constructor.

readonly attribute Promise closed

Detects when the TCP server socket been closed, either cleanly by the client application calling `close()`) or through an error situation. Returns the `closedPromise` that was created in the `TCPSocket` constructor.

Promise listen()

Listens for incoming connection attempts on the specified port and address. Returns the `connectionPromise`, which is for a succeful connection resolved with the `TCPSocket` object for the accepted TCP connection and rejected with DOMException "`NetworkError`" if there is an error on an incoming connection attempt.

Promise close()

Closes the TCP server socket. If `listen()` has been called the listening for incoming connections is stopped but existing TCP connections are kept open. Returns the `closedPromise` that was created in the `TCPServerSocket` constructor.

When the TCPServerSocket constructor is invoked, the User Agent MUST run the following steps:

1. Create a new TCPServerSocket object ("`myServerSocket`").
2. If the `options` argument's `localAddress` member is absent the server will accept connections directed to any IPv4 address and the `localAddress` attribute is set to `null`. Otherwise, if the requested local address is a valid IPv4/6 address for a local interface on the device bind the server socket to this local IPv4/6 address and set the `localAddress` attribute to this address. Else, if the `localAddress` member is present but it is not a valid local IPv4/6 address for a local interface on the device, throw DOMException `InvalidAccessError` and abort the remaining steps.
3. If the `options` argument's `localPort` member is absent then bind the socket to an ephemeral local port decided by the system and set the `localPort` attribute to `null`. Otherwise, bind the socket to the requested local port and set the `localPort` attribute to the local port that the socket is bound to.
4. Set the `addressReuse` attribute to the value of the `options` argument's `addressReuse` member if it is present or to `true` if the `options` argument's `addressReuse` member is not present.
5. Set the `myServerSocket.readyState` attribute to "opening".
6. Create a new promise, "`openedPromise`", and store it so it can later be returned by the `opened` property.

7. Create a new promise, "`closedPromise`", and store it so it can later be returned by the `closed` property and the `close` method.
8. Return the newly created TCPServerSocket object to the application.

The *`close`* method when invoked MUST run the following steps:

1. If a TCP connection setup is in progress the connection setup is finalized according to the descriptions below.
2. Stop listening to further connection attempts from clients.
3. Set the `myServerSocket.readyState` attribute to "closed".
4. Resolve `closedPromise` with `undefined`.

The *`listen`* method when invoked MUST run the following steps:

1. If `myServerSocket.readyState` attribute is"closed" then throw DOMException "`InvalidStateError`" and abort the remaining steps.
2. Create a new promise, "`connectionPromise`".
3. Start listening for connections on the specified local port and address. Return `connectionPromise`.

When a new TCP server socket has successfully been created the user agent MUST run the following steps:

1. Change the `myServerSocket.readyState` attribute's value to "open".
2. Resolve `openedPromise` with `undefined`.

When the attempt to create a new TCP server socket (`myServerSocket.readyState` is "opening") has failed the user agent MUST run the following steps:

1. Change the `myServerSocket.readyState` attribute's value to "closed".
2. Reject `openedPromise` with DOMException "`NetworkError`".

When there is an error on an established TCP server socket (`myServerSocket.readyState` is "open"), e.g. loss of network contact, the user agent MUST run the following steps:

1. Change the `myServerSocket.readyState` attribute's value to "closed".
2. Reject `closedPromise` with DOMException "`NetworkError`".

Upon a new successful connection to the TCP server socket the user agent MUST run the following steps:

1. Let *socket* be a new instance of TCPSocket.
2. Set the `remoteAddress` attribute of *socket* to the IPv4/6 address of the peer.
3. Set the `remotePort` attribute of *socket* to the source port of the of the peer.
4. Set the `localAddress` attribute of *socket* to the used local IPv4/6 address.

5. Set the `localPort` attribute of *socket* to the used local source port.
6. Set the `readyState` attribute of *socket* to "open".
7. Set the `bufferedAmount` attribute of *socket* to 0.
8. Resolve `connectionPromise` with *socket* as argument.

Upon a new connection attempt to the TCP server socket that can not be served, e.g. due to max number of open connections, the user agent MUST run the following steps:

1. Reject `connectionPromise` with DOMException `"NetworkError"`.

J.7. DICTIONARY UDPMESSAGE

The UDPMessage dictionary represents UDP data including address and port of the remote peer. The field data is mandatory but remoteAddress and remotePort are optional.

ArrayBuffer data
Received UDP data or UDP data to send.

DOMString remoteAddress
The address of the remote machine.

unsigned short remotePort
The port of the remote machine.

J.8. DICTIONARY UDPOPTIONS

States the options for the UDPSocket. An instance of this dictionary can optionally be used in the constructor of the UDPSocket object, where all fields are optional.

DOMString localAddress
The IPv4/6 address of the local interface, e.g. wifi or 3G, that the UDPSocket object is bound to. If the field is omitted, the user agent binds the socket to an IPv4/6 address based on the routing table and possibly a preselect default local interface to use for the selected `remoteAddress` if this member is present. Else the UDPSocket is unbound to a local interface.

unsigned short localPort
The local port that the UDPSocket object is bound to. If the the field is omitted, the user agent binds the socket to a an ephemeral local port decided by the system.

DOMString remoteAddress
When present the default remote host name or IPv4/6 address that is used for subsequent send() calls.

unsigned short remotePort
When present the default remote port that is used for subsequent send() calls.

boolean addressReuse
`true` allows the socket to be bound to a local address/port pair that already is in use. Default is `true`.

boolean loopback
Only applicable for multicast.`true` means that sent multicast data is looped back to the sender. Default is `false`.

J.9. DICTIONARY TCPOPTIONS

States the options for the TCPSocket. An instance of this dictionary can optionally be used in the constructor of the TCPSocket object, where all fields are optional.

DOMString localAddress
The IPv4/6 address of the local interface, e.g. wifi or 3G, that the TCPSocket object is bound to. If the field is omitted, the user agent binds the socket to an IPv4/6 address based on the routing table and possibly a preselect default local interface to use for the selected `remoteAddress`.

unsigned short localPort
The local port that the TCPSocket object is bound to. If the the field is omitted, the user agent binds the socket to an ephemeral local port decided by the system.

boolean addressReuse
`true` allows the socket to be bound to a local address/port pair that already is in use. Default is `true`.

boolean noDelay
`true` if the Nagle algorithm for send coalescing, [[!NAGLE]], is disabled. Default is `true`.

boolean useSecureTransport
`true` if socket uses SSL or TLS. Default is `false`.

Use of secure transport needs more investigation

J.10. DICTIONARY TCPSERVEROPTIONS

States the options for the TCPServerSocket. An instance of this dictionary can optionally be used in the constructor of the TCPServerSocket object, where all fields are optional.

DOMString localAddress
The IPv4/6 address of the interface, e.g. wifi or 3G, that the TCPServerSocket object is bound to. If the field is omitted, the user agent binds the server socket to the IPv4/6 address of the default local interface.

unsigned short localPort
The local port that the TCPServerSocket object is bound to. If the the field is omitted, the user agent binds the socket to an ephemeral local port decided by the system.

boolean addressReuse
`true` allows the socket to be bound to a local address/port pair that already is in use. Default is `true`.

boolean useSecureTransport
`true` if socket uses SSL or TLS. Default is `false`.

Use of secure transport needs more investigation

J.11. ENUMS

J.11.1. SocketReadyState

opening
The socket is in opening state, i.e. availability of local address/port is being checked, network status is being checked, etc. For TCP a connection with a remote peer has not yet been established.

open
The socket is ready to use to send and received data. For TCP a connection with a remote peer has been established.

closing
Only used for TCP sockets. The TCP connection is going through the closing handshake, or the close() method has been invoked.

closed
The socket is closed and can not be use to send and received data. For TCP the connection has been closed or could not be opened.

halfclosed
Only used for TCP sockets. The TCP connection has been "halfclosed" by the application, which means that it is not possible to send data but it is still possible to receive.

J.12. ACKNOWLEDGEMENTS

Many thanks to Domenic Denicola, Marcos Caceres, Jonas Sicking, Ke-Fong Lin and Alexandre Morgaut for reviewing the specification and providing very valuable feedback. Also thanks to Sony colleagues Anders Edenbrandt, Anders Isberg and Björn Isaksson for sharing their experience on socket APIs and providing support.

K. WEB BLUETOOTH

W3C Community Group Editor's Draft 09 October 2014

Latest editor's draft:
http://webbluetoothcg.github.io/web-bluetooth/

Editors:
See contributors on GitHub

This document describes an API to discover and communicate with devices over the Bluetooth 4 wireless standard using the Generic Attribute Profile (GATT).

Changes to this document may be tracked at https://github.com/WebBluetoothCG/web-bluetooth/commits/gh-pages.

K.1. INTRODUCTION

Bluetooth is a standard for short-range wireless communication between devices. Bluetooth "Classic" (BR/EDR) defines a set of binary protocols and supports speeds up to about 24Mbps. Bluetooth 4.0 introduced a new "Low Energy" mode known as "Smart", BTLE, or just LE which is limited to about 1Mbps but allows devices to leave their transmitters off most of the time. BTLE provides most of its functionality through key/value pairs provided by the Generic Attribute Profile (GATT).

BTLE defines multiple roles that devices can play. The *Broadcaster* and *Observer* roles are for transmitter- and receiver-only applications, respectively. The *Peripheral* role is able to receive a single connection. The *Central* role supports multiple connections, and is responsible for creating any connections to Peripheral devices.

GATT further defines *Client* and *Server* roles, which are orthogonal to the Peripheral and Central BTLE roles. GATT allows Servers to expose Services, whose type is identified by a UUID ([[!RFC4122]]). A *Service* exposes a collection of included Services and Characteristics. Each *Characteristic* has a type named by a UUID and exposes a value as an array of bytes, some properties to describe how the value can be used, and a collection of Descriptors. Each *Descriptor* has a type named by a UUID and contains related information about the Characteristic Value.

Despite being designed to support BTLE transport, the GATT protocol can also run over BR/EDR transport. Both support advertising GATT services without a connection: BTLE through advertising packets, and BR/EDR through the extended inquiry response.

The first version of this specification focuses on the Bluetooth 4 GATT protocol in the Central and Client roles, over either a BR/EDR or LE connection. While this specification cites the [[BLUETOOTH41]] specification, it intends to also support communication with devices that only implement Bluetooth 4.0.

K.2. SECURITY AND PRIVACY CONSIDERATIONS

K.3. DEVICE ACCESS IS POWERFUL

When a website requests access to devices using `requestDevice`, it gets the ability to access all GATT services mentioned in the call. The UA MUST inform the user what capabilities these services give the website before asking which devices to entrust to it. If any services in the list aren't known to the UA, the UA MUST assume they give the site complete control over the device and inform the user of this risk. The UA MUST also allow the user to inspect what sites have access to what devices and revoke these pairings.

The UA MUST NOT allow the user to pair entire classes of devices with a website. It is possible to construct a class of devices for which each individual device sends the same Bluetooth-level identifying information. UAs are not required to attempt to detect this sort of forgery and MAY let a user pair this pseudo-device with a website.

To help ensure that only the entity the user approved for access actually has access, this specification requires that only authenticated environments can access Bluetooth devices (requestDevice).

K.4. ATTACKS ON DEVICES

We expect that many devices are vulnerable to unexpected data delivered to their radio. In the past, these devices had to be exploited one-by-one, but this API makes it plausible to conduct large-scale attacks. This specification takes several approaches to make such attacks more difficult:

- Pairing individual devices instead of device classes requires at least a user action before a device can be exploited.
- Constraining access to GATT, as opposed to generic byte-stream access, denies malicious websites access to most parsers on the device.

  On the other hand, GATT's Characteristic and Descriptor values are still byte arrays, which may be set to lengths and formats the device doesn't expect. UAs are encouraged to validate these values when they can.

- This API never exposes Bluetooth addressing, data signing or encryption keys ([[!BLUETOOTH41]] Volume 3 Part H Section 2.4.1) to websites.

This makes it more difficult for a website to predict the bits that will be sent over the radio, which blocks packet-in-packet injection attacks. Unfortunately, this only works over encrypted links, which not all BTLE devices are required to support.

UAs can also take further steps to protect their users:

- A web service may collect lists of malicious websites and vulnerable devices. UAs can deny malicious websites access to any device and any website access to vulnerable devices.

K.5. BLUETOOTH DEVICE IDENTIFIERS

Each Bluetooth BR/EDR device has a unique 48-bit MAC address known as the *BD_ADDR* ([[!BLUETOOTH41]] 3.C.15). Each Bluetooth LE device has at least one of a Public Device Address and a Static Random Address. The *Public Device Address* is a MAC address ([[!BLUETOOTH41]] 6.B.1.3). The *Static Random Address* may be regenerated on each restart ([[!BLUETOOTH41]] 3.C.10.8). A BR/EDR/LE device will use the same value for the BD_ADDR and the Public Device Address ([[!BLUETOOTH41]] 2.E.7.4.6).

An LE device may also have a unique, 128-bit *Identity Resolving Key (IRK)* ([[!BLUETOOTH41]] 3.H.2.4.2.1), which is sent to trusted devices during the bonding process. To avoid leaking a persistent identifier, an LE device may scan and advertise using a random *Resolvable Private Address* or *Non-Resolvable Private Address* instead of its Static or Public Address. These are regenerated periodically (approximately every 15 minutes), but a bonded device can check whether one of its stored IRKs matches any given Resolvable Private Address ([[!BLUETOOTH41]] 3.C.10.8.2.3).

Each Bluetooth device also has a human-readable *Device Name* (([[!BLUETOOTH41]] 3.C.3.2.2). These aren't guaranteed to be unique, but may well be, depending on the device type.

K.6. IDENTIFIERS FOR REMOTE BLUETOOTH DEVICES

If a website can retrieve any of the persistent device IDs, these can be used, in combination with a large effort to catalog ambient devices, to discover a user's location. A device ID can also be used to identify that a user who pairs two different websites with the same Bluetooth device is a single user. On the other hand, many GATT services are available that could be used to fingerprint a device, and a device can easily expose a custom GATT service to make this easier.

Because it would be so easy to work around an attempt to block device identification, this spec doesn't try to do so.

K.7. THE UA'S BLUETOOTH ADDRESS

In BR/EDR mode, or in LE mode during active scanning without the Privacy Feature, the UA broadcasts its persistent ID to any nearby Bluetooth radio. This makes it easy to scatter hostile devices in an area and track the UA. As of 2014-08, few or no platforms document that they implement the Privacy Feature, so despite this spec recommending it, few UAs are likely to use it. This spec does require a user gesture for a website to trigger a scan, which reduces the frequency of scans some, but it would still be better for more platforms to expose the Privacy Feature.

K.8. DEVICE DISCOVERY

The UA MUST maintain an *allowed devices list* for each origin, storing a set of Bluetooth devices the origin is allowed to access. For each device in the allowed devices list for an origin, the UA MUST maintain an *allowed services list* consisting of UUIDs for GATT Primary Services the origin is allowed to access on the device. The UA MAY remove devices from the allowed devices list at any time based on signals from the user. For example, if the user chooses not to remember access, the UA might remove a device when the tab that was granted access to it is closed. Or the UA might provide a revocation UI that allows the user to explicitly remove a device even while a tab is actively using that device. If a device is removed from this list while a Promise is pending to do something with the device, it MUST be treated the same as if the device moved out of Bluetooth range.

Promise<BluetoothDevice> requestDevice(sequence<BluetoothScanFilter> filters, optional RequestDeviceOptions options)

When this method is invoked, the UA MUST run the following steps:

1. Return a new Promise, but continue running these steps asynchronously.
2. If the global environment is not an authenticated environment, reject the Promise with a SecurityError and abort these steps.
3. If the algorithm is not allowed to show a popup, reject the Promise with a SecurityError and abort these steps.
4. Scan for devices with the union of all `services` sequences in *`filters`* as the *set of Service UUIDs*
5. Remove devices from the result of the scan if they do not match a filter in *filters*.
6. Display a prompt to the user requesting that the user specify some devices from the result of the scan. The UA SHOULD show the user the human-readable name of each device. If this name is not available because the UA's Bluetooth system doesn't support privacy-enabled scans, the UA SHOULD allow the user to indicate interest and then perform a privacy-disabled scan to retrieve the name.

The UA MAY allow the user to select a nearby device that does not match *filters*.

7. Wait for the user to have made their selection.
8. If the user cancels the prompt, reject the Promise with a NotFoundError and abort these steps.
9. Record the selected device in the origin's allowed devices list and the union of the service UUIDs from `filters` and `options.optionalServices` in the device and origin's allowed services list.
10. Connect to the device. ([[BLUETOOTH41]] 3.G.6.2.1) If the connection fails, reject the Promise with a NetworkError and abort these steps.
11. Resolve the Promise with a BluetoothDevice instance representing the selected device.

sequence<BluetoothServiceUuid> services
A list of Service UUIDs that a device must support to match this filter.

sequence<BluetoothServiceUuid> optionalServices = []
A list of Service UUIDs that aren't required for the website to use a device, but that the website can take advantage of if they're present.

A device *matches a filter filter* if

- `*filter*.services` is not empty and
- the UA has received advertising data, an extended inquiry response, or a service discovery response indicating that the device supports each of the Service UUIDs included in `*filter*.services` as a primary (vs included) service.

The list of Service UUIDs that a device advertises might not include all the UUIDs the device supports. The advertising data does specify whether this list is complete. If a website filters for a UUID that a nearby device supports but doesn't advertise, that device might not be included in the list of devices presented to the user. The UA would need to connect to the device to discover the full list of supported services, which can impair radio performance and cause delays, so this spec doesn't require it.

To *scan for devices* with an optional *set of Service UUIDs*, defaulting to the set of all UUIDs, the UA MUST perform the following steps:

1. If the UA has scanned for devices recently TODO: Nail down the amount of time. with a set of UUIDs that was a superset of the UUIDs for the current scan, then the UA MAY return the result of that scan and abort these steps.
2. Let *result* be a set of Bluetooth devices, initially empty.
3. If the UA supports the LE transport:

1. The UA MUST perform the *General Discovery Procedure* ([[BLUETOOTH41]] 3.C.9.2.6). The UA SHOULD enable the *Privacy Feature* ([[BLUETOOTH41]] 3.C.10.7).

   Both *passive scanning* ([[!BLUETOOTH41]] 6.B.4.4.3.1) and the Privacy Feature avoid leaking the unique, immutable device ID. We ought to require UAs to use either one, but none of the OS APIs appear to expose either. Bluetooth also makes it hard to use passive scanning since it doesn't require Central devices to support the *Observation Procedure* ([[!BLUETOOTH41]] 3.C.9.1.2).

2. For each *discovered LE device* ([[BLUETOOTH41]] 3.C.9.2.6), if the complete device name was not acquired during the General Discovery Procedure, the UA SHOULD perform the *Name Discovery Procedure* ([[BLUETOOTH41]] 3.C.9.2.7).
3. For each discovered LE device, if the advertised Service UUIDs ([[BLUETOOTH-SUPPLEMENT4]], A.1.1) have a non-empty intersection with the *set of Service UUIDs*, add the device to *result*.
4. For each discovered LE device, the UA MAY connect to the device and
   - Use *Attribute Caching* ([[BLUETOOTH41]] 3.G.2.5.2) and the *Service Changed characteristic* ([[BLUETOOTH41]] 3.G.7.1) to recall and validate the supported Service UUIDs from a previous connection,
   - *Discover Primary Service by Service UUID* ([[BLUETOOTH41]] 3.G.4.4.2) for each UUID in the *set of Service UUIDs*, or
   - *Discover All Primary Services* ([[BLUETOOTH41]] 3.G.4.4.1).

   If one of the discovered services is in the *set of Service UUIDs*, add the device to *result*.

   Connecting to every nearby device to discover services costs power and can slow down other use of the Bluetooth radio. UAs should only discover extra services on a device if they have some reason to expect that device to be interesting.

   UAs should also help developers avoid relying on this extra discovery behavior. For example, say a developer has previously connected to a device, so the UA knows the device's full set of supported services. If this developer then filters using a non-advertised UUID, the dialog they see may include this device, even if the filter would likely exclude the device on users' machines. The UA could provide a developer option to warn when this happens or to include only advertised services in matching filters.

4. If the UA supports the BR/EDR transport:
    1. The UA MUST perform the *Device Discovery* procedure ([[BLUETOOTH41]] 3.C.6.4).

       All forms of BR/EDR inquiry/discovery appear to leak the unique, immutable device address.

    2. For each *discovered BR/EDR device* ([[BLUETOOTH41]] 3.C.6.4.3), if the Service UUIDs ([[BLUETOOTH-SUPPLEMENT4]], A.1.1) in the *Extended Inquiry Response* ([[BLUETOOTH41]] 1.A.4.2.1.1.1) have a non-empty intersection with the *set of Service UUIDs*, add the device to *result*.

       There is no way to distinguish GATT from non-GATT services in the Extended Inquiry Response. If a site filters to the UUID of a non-GATT service, the user may be able to select a device for the result of `requestDevice` that this API provides no way to interact with.

    3. For each discovered BR/EDR device, the UA MAY connect to the device and
        - Use Attribute Caching and the Service Changed characteristic to recall and validate the supported Service UUIDs from a previous connection,
        - Discover Primary Service by Service UUID for each UUID in the *set of Service UUIDs*,
        - Discover All Primary Services, or
        - use the Service Discovery Protocol (SDP) to find GATT Services with UUIDs in the *set of Service UUIDs*. ([[BLUETOOTH41]] 3.B.2.5)

       If one of the discovered services is in the *set of Service UUIDs*, add the device to *result*.

       See the note for discovered LE devices.

5. Return *result* from the scan.

We need a way for a site to register to receive an event when an interesting device comes within range.

K.9. BLUETOOTHDEVICE

Represents a known Bluetooth device.

readonly attribute DOMString instanceId

Returns the opaque identifier assigned to this device. This identifier MUST uniquely identify a device to the extent that the UA can determine that two Bluetooth connections are to the same device. For example, a paired device MUST keep the same `instanceId` across a change to its Resolvable

Private Address. On the other hand, a device using Non-Resolvable Private Addresses might not keep the same `instanceId`. The UA MAY use different identifiers for the same device on two different origins. The UA MAY create a new identifier for a known device on an origin if the user revokes and re-grants access to that device. The UA MAY re-use identifiers for existing devices on an origin if the user clears the origin's data.

readonly attribute DOMString? name
The human-readable name of the device.

readonly attribute long? deviceClass
The class of the device, a bit-field defined by [[!BLUETOOTH-ASSIGNED-BASEBAND]].

readonly attribute VendorIdSource? vendorIdSource
The Vendor ID Source field in the pnp_id characteristic in the device_information service.

readonly attribute long? vendorId
The 16-bit Vendor ID field in the pnp_id characteristic in the device_information service.

readonly attribute long? productId
The 16-bit Product ID field in the pnp_id characteristic in the device_information service.

readonly attribute long? productVersion
The 16-bit Product Version field in the pnp_id characteristic in the device_information service.

readonly attribute boolean? paired
Indicates whether or not the device is paired with the system.

readonly attribute boolean? connected
Indicates whether the device is currently connected to the system.

readonly attribute sequence<UUID>? uuids
UUIDs of protocols, profiles and services advertised by the device. For Low Energy devices, this list is obtained from AD and GATT primary services.

Promise<void> connect()
Establishes a connection to the device.

Promise<void> disconnect()
Closes the site's connection to the device. Note that this will not always destroy the physical link itself, since there may be other sites with open connections.

Promise<BluetoothGATTService> getPrimaryService(BluetoothServiceUuid service)
: Returns a promise that is asynchronously resolved with the first primary GATT service on the remote device whose UUID is *service* and whose UUID is in the allowed services list for this device and origin. If there is no such service, resolves the promise with `null`.

Promise<sequence<BluetoothGATTService>> getPrimaryServices()
: Returns a promise that is asynchronously resolved with a sequence of all the primary GATT services on the remote device whose UUIDs are in the allowed services list for this device and origin.

Promise<sequence<BluetoothGATTService>> getPrimaryServices(BluetoothServiceUuid service)
: Returns `this.getPrimaryServices([service])`

Promise<sequence<BluetoothGATTService>> getPrimaryServices(sequence<BluetoothServiceUuid> services)
: Returns a promise that is asynchronously resolved with a sequence of all the primary GATT services on the remote device with UUIDs in both *services* and the allowed services list for this device and origin.

Allocation authorities for Vendor IDs.

bluetooth

usb

K.10. GATT INTERACTION

K.11. BLUETOOTHGATTSERVICE

BluetoothGATTService represents a GATT Service within a Bluetooth Peripheral, a collection of characteristics and relationships to other services that encapsulate the behavior of part of a device.

readonly attribute UUID uuid
: The UUID of the service, e.g. `UUID.parse('0000180d-0000-1000-8000-00805f9b34fb')`.

readonly attribute boolean isPrimary
: Indicates whether the type of this service is primary or secondary.

readonly attribute DOMString instanceId
: Returns the opaque identifier assigned to this service, which can be used distinguish between multiple primary services with the same UUID in a single device or multiple included services with the same UUID in a single primary service. This identifier MUST be unique among all services accessible by this website. This identifier MUST continue referring to the same Service until either an `onServiceRemoved` event is delivered referring

to this BluetoothGATTService or if this service's device is not `paired`, it is disconnected.

readonly attribute BluetoothDevice device
The BluetoothDevice representing the remote peripheral that the GATT service belongs to.

Promise<BluetoothGATTCharacteristic> getCharacteristic(BluetoothCharacteristicUuid characteristic)
Returns a promise that is asynchronously resolved with the first GATT characteristic within this Service whose UUID is *characteristic*.

Promise<sequence<BluetoothGATTCharacteristic>> getCharacteristics()
Returns a promise that is asynchronously resolved with a sequence of all the GATT characteristics within this Service.

Promise<sequence<BluetoothGATTCharacteristic>> getCharacteristics(BluetoothCharacteristicUuid characteristic)
Returns `this.getCharacteristics([characteristic])`

Promise<sequence<BluetoothGATTCharacteristic>> getCharacteristics(sequence<BluetoothCharacteristicUuid> characteristics)
Returns a promise that is asynchronously resolved with a sequence of all the GATT characteristics within this Service with UUIDs in *characteristics*.

Promise<BluetoothGATTService> getIncludedService(BluetoothServiceUuid service)
Returns a promise that is asynchronously resolved with the first GATT included service (in the order returned by the Find Included Services procedure: [[BLUETOOTH41]] 3.G.4.5.1) within this Service whose UUID is *service*.

Promise<sequence<BluetoothGATTService>> getIncludedServices()
Returns a promise that is asynchronously resolved with a sequence of all the GATT included services within this Service.

Promise<sequence<BluetoothGATTService>> getIncludedServices(BluetoothServiceUuid service)
Returns `this.getIncludedServices([service])`

Bluetooth's procedure for finding included services (3.G.4.5) doesn't include a way to optimize for a single UUID. Maybe we should omit this and the next overloads to avoid implying that they're more efficient than the no-argument version.

Promise<sequence<BluetoothGATTService>> getIncludedServices(sequence<BluetoothServiceUuid> services)

Returns a promise that is asynchronously resolved with a sequence of all the GATT included services within this Service with UUIDs in *services*.

K.12. BLUETOOTHGATTCHARACTERISTIC

BluetoothGATTCharacteristic represents a GATT Characteristic, which is a basic data element that provides further information about a peripheral's service.

readonly attribute UUID uuid

The UUID of the characteristic, e.g. `UUID.parse('00002a37-0000-1000-8000-00805f9b34fb')`.

readonly attribute BluetoothGATTService service

The GATT service this characteristic belongs to.

readonly attribute CharacteristicProperty[] properties

The properties of this characteristic.

readonly attribute DOMString instanceId

Returns the opaque identifier assigned to this characteristic, which can be used distinguish between multiple characteristics with the same UUID in a single service. This identifier MUST be unique among all characteristics accessible by this website. This identifier MUST continue referring to the same Characteristic until either an `onServiceRemoved` or `onServiceChanged` event is delivered referring to the BluetoothGATTService of this characteristic or if this characteristic's BluetoothDevice is not `paired`, it is disconnected.

readonly attribute ArrayBuffer? value

The currently cached characteristic value. This value gets updated when the value of the characteristic is read or updated via a notification or indication.

Promise<BluetoothGATTDescriptor> getDescriptor(BluetoothDescriptorUuid descriptor)

Returns a promise that is asynchronously resolved with the first GATT descriptor within this Characteristic whose UUID is *descriptor*.

Promise<sequence<BluetoothGATTDescriptor>> getDescriptors()

Returns a promise that is asynchronously resolved with a sequence of all the GATT descriptors within this Characteristic.

Promise<sequence<BluetoothGATTDescriptor>> getDescriptors(BluetoothDescriptorUuid descriptor)

Returns `this.getDescriptors([descriptor])`

Promise<sequence<BluetoothGATTDescriptor>> getDescriptors(sequence<BluetoothDescriptorUuid> descriptors)

Returns a promise that is asynchronously resolved with a sequence of all the GATT descriptors within this Characteristic with UUIDs in *descriptors*.

Promise<ArrayBuffer> readValue()

The UA MUST return a new promise and asynchronously read the value of this characteristic resolving the promise.

Promise<void> writeValue()

Write the value of a specified characteristic from a remote peripheral.

ArrayBuffer value

The value that should be sent to the remote characteristic as part of the write request.

Promise<void> startNotifications()

The UA MUST return a new Promise and asynchronously enable notifications on this characteristic resolving the promise. See for details of receiving notifications.

Promise<void> stopNotifications()

The UA MUST return a new Promise and asynchronously disable notifications on this characteristic resolving the promise.

To *read the value of a BluetoothGATTCharacteristic* resolving a promise, the UA MUST:

1. Let *characteristic* be the Characteristic that the BluetoothGATTCharacteristic represents.
2. If the `Read` bit is not set in *characteristic*'s properties ([[!BLUETOOTH41]] 3.G.3.3.1.1), reject the promise with a NotSupportedError and abort these steps.
3. Use any combination of the sub-procedures in the Characteristic Value Read procedure ([[!BLUETOOTH41]] 3.G.4.8) to retrieve the value of *characteristic*. Use the Bluetooth error recovery procedure with the promise. If this fails, abort these steps.
4. Create an `ArrayBuffer` holding the retrieved value, and assign it to the BluetoothGATTCharacteristic's `value` field.
5. Fire an event named `characteristicvaluechanged` with its `bubbles` attribute initialized to `true` at the BluetoothGATTCharacteristic.

For each known GATT Characteristic, the UA MUST maintain an *active notification context set* of BluetoothInteraction objects. This is a single set for the whole UA, pointing to the `navigator.bluetooth` object for each separate script execution environment that has registered for notifications.

To *enable notifications* on a BluetoothGATTCharacteristic resolving a promise, the UA MUST:

1. Let *characteristic* be the GATT Characteristic that the BluetoothGATTCharacteristic represents.
2. If neither of the `Notify` or `Indicate` bits are set in *characteristic*'s properties ([[!BLUETOOTH41]] 3.G.3.3.1.1), reject the promise with a NotSupportedError and abort these steps.
3. If the active notification context set contains `navigator.bluetooth`, resolve the promise and abort these steps.
4. Ensure that one of the `Notification` or `Indication` bits in *characteristic*'s Client Characteristic Configuration descriptor ([[!BLUETOOTH41]] 3.G.3.3.3.3) is set, matching the constraints in *characteristic*'s properties. The UA SHOULD avoid setting both bits, and MUST deduplicate value-change events if both bits are set. Use the Bluetooth error recovery procedure with the promise. If this fails, abort these steps.
5. Add `navigator.bluetooth` to the active notification context set.
6. Resolve the promise.

After notifications are enabled, the resulting value-change events won't be delivered until after the current microtask checkpoint. This allows a developer to set up handlers in the `.then` handler of the result promise.

To *disable notifications* on a BluetoothGATTCharacteristic resolving a promise, the UA MUST:

1. Let *characteristic* be the GATT Characteristic that the BluetoothGATTCharacteristic represents.
2. If the active notification context set contains `navigator.bluetooth`, remove it.
3. If the active notification context set became empty, the UA SHOULD clear the `Notification` and `Indication` bits in *characteristic*'s Client Characteristic Configuration descriptor ([[!BLUETOOTH41]] 3.G.3.3.3.3).
4. Queue a task to resolve the promise.

Queuing a task to resolve the promise ensures that no value change events due to notifications arrive after the promise resolves.

K.13. BLUETOOTHGATTDESCRIPTOR

BluetoothGATTDescriptor represents a GATT Descriptor, which provides further information about a Characteristic's value.

readonly attribute UUID uuid
The UUID of the characteristic descriptor, e.g. `'00002902-0000-1000-8000-00805f9b34fb'`.

readonly attribute BluetoothGATTCharacteristic characteristic
The GATT characteristic this descriptor belongs to.

readonly attribute DOMString instanceId

Returns the opaque identifier assigned to this descriptor, which can be used distinguish between multiple descriptors with the same UUID in a single characteristic. This identifier MUST be unique among all descriptors accessible by this website. This identifier MUST continue referring to the same Descriptor until either an `onServiceRemoved` or `onServiceChanged` event is delivered referring to the BluetoothGATTService of this descriptor or if this descriptor's device is not `paired`, it is disconnected. Use the instance ID to distinguish between descriptors from a peripheral with the same UUID and to make function calls that take in a descriptor identifier. Present, if this instance represents a remote characteristic.

readonly attribute ArrayBuffer? value

The currently cached descriptor value. This value gets updated when the value of the descriptor is read.

Promise<ArrayBuffer> readValue()

Retrieve the value of this descriptor from a remote peripheral. Updates the descriptor's `value` field to hold the result of the read request and resolves the promise with the same ArrayBuffer.

Promise<void> writeValue()

Write the value of a specified characteristic descriptor from a remote peripheral.

ArrayBuffer value

The value that should be sent to the remote descriptor as part of the write request.

K.14. OBJECT AND UUID LOOKUP ON `NAVIGATOR.BLUETOOTH`

readonly attribute BluetoothUuids uuids

Promise<BluetoothGATTService> getService()

Get the GATT service with the given instance ID.

DOMString serviceInstanceId

The instance ID of the requested GATT service.

Promise<BluetoothGATTCharacteristic> getCharacteristic()

Get the GATT characteristic with the given instance ID that belongs to the given GATT service, if the characteristic exists.

DOMString characteristicInstanceId

The instance ID of the requested GATT characteristic.

Promise<BluetoothGATTDescriptor> getDescriptor()

Get the GATT characteristic descriptor with the given instance ID.

DOMString descriptorInstanceId

The instance ID of the requested GATT characteristic descriptor.

K.15. EVENTS

K.16. BLUETOOTH TREE

`navigator.bluetooth` and objects implementing the BluetoothDevice, BluetoothGATTService, BluetoothGATTCharacteristic, or BluetoothGATTDescriptor interface participate in a tree, simply named the *Bluetooth tree*.

- The children of `navigator.bluetooth` are the BluetoothDevice objects representing devices on the origin's allowed devices list, in an unspecified order.
- The children of a BluetoothDevice are the BluetoothGATTService objects representing Primary and Secondary Services on its GATT Server whose UUIDs are on the origin and device's allowed services list. The order of the primary services MUST be consistent with the order returned by the Discover Primary Service by Service UUID procedure ([[!BLUETOOTH41]] 3.G.4.4.2), but secondary services and primary services with different UUIDs may be in any order.
- The children of a BluetoothGATTService are the BluetoothGATTCharacteristic objects representing its Characteristics. The order of the characteristics MUST be consistent with the order returned by the Discover Characteristics by UUID procedure ([[!BLUETOOTH41]] 3.G.4.6.2), but characteristics with different UUIDs may be in any order.
- The children of a BluetoothGATTCharacteristic are the BluetoothGATTDescriptor objects representing its Descriptors in the order returned by the Discover All Characteristic Descriptors procedure ([[!BLUETOOTH41]] 3.G.4.7.1)

K.17. EVENT TYPES

characteristicvaluechanged
: Fired on a BluetoothGATTCharacteristic when its value changes, either as a result of a read request, or a value change notification/indication.

serviceadded
: Fired on a new BluetoothGATTService when it has been discovered on a remote device, just after it is added to the Bluetooth tree.

servicechanged
: Fired on a BluetoothGATTService when its state changes. This involves any characteristics and/or descriptors that get added or removed from the service, as well as Service Changed indications ([[!BLUETOOTH41]] 3.G.7.1) from the remote device.

serviceremoved
: Fired on a BluetoothGATTService when it has been removed from its device, just before it is removed from the Bluetooth tree.

K.18. RESPONDING TO NOTIFICATIONS AND INDICATIONS

When the UA receives a Bluetooth Notification ([[!BLUETOOTH41]] 3.G.4.10) or Indication ([[!BLUETOOTH41]] 3.G.4.11) for a Characteristic, it must perform the following steps:

1. For each *bluetoothGlobal* in the active notification context set for the Characteristic, queue a task on the event loop of the script settings object of *bluetoothGlobal* to do the following steps:
    1. Let *characteristicObject* be the BluetoothGATTCharacteristic in the Bluetooth tree rooted at *bluetoothGlobal* that represents the Characteristic.
    2. Set `characteristicObject.value` to a new `ArrayBuffer` holding the new value of the Characteristic.
    3. Fire an event named `characteristicvaluechanged` with its `bubbles` attribute initialized to `true` at *characteristicObject*.

K.19. RESPONDING TO SERVICE CHANGES

The Bluetooth Attribute Caching system ([[!BLUETOOTH41]] 3.G.2.5.2) allows clients to track changes to Services, Characteristics, and Descriptors. Before discovering any of these entities for the purpose of exposing them to a web page the UA MUST subscribe to Indications from the Service Changed characteristic ([[!BLUETOOTH41]] 3.G.7.1), if it exists. When the UA receives an Indication on the Service Changed characteristic, it MUST perform the following steps.

1. Let *removedEntities* be the list of entities in the range indicated by the Service Changed characteristic that the UA had discovered before the Indication.
2. Use the Primary Service Discovery, Relationship Discovery, Characteristic Discovery, and Characteristic Descriptor Discovery procedures ([[!BLUETOOTH41]] 3.G 4.4, 4.5, 4.6, and 4.7) to re-discover entities in the range indicated by the Service Changed characteristic. The UA MAY skip discovering all or part of the indicated range if it can prove that the results of that discovery could not affect the events fired below.
3. Let *addedEntities* be the list of entities discovered in the previous step.
4. If an entity with the same definition, ignoring Characteristic and Descriptor values, appears in both *removedEntities* and *addedEntities*, remove it from both.
5. Let *changedServices* be a set of Services, initially empty.
6. If the same Service appears in both *removedEntities* and *addedEntities*, remove it from both, and add it to *changedServices*. Services with different UUIDs or Primary/Secondary status MUST NOT be considered "the same", but this specification says nothing else about determining identity.

7. For each Characteristic and Descriptor in *removedEntities* and *addedEntities*, remove it from its original list, and add its parent Service to *changedServices*. After this point, *removedEntities* and *addedEntities* contain only Services.
8. If a Service in *addedEntities* would not have been returned to any script execution environment if it had existed at the time of any previous call to `getPrimaryService`, `getPrimaryServices`, `getIncludedService`, or `getIncludedServices`, the UA MAY remove the Service from *addedEntities*.
9. Let *changedDevices* be the set of Bluetooth devices that contain any Service in *removedEntities*, *addedEntities*, and *changedServices*.
10. For each script execution environment that is connected to a device in *changedDevices*, queue a task on its event loop to do the following steps:
    1. For each Service in *removedEntities*, fire an event named `serviceremoved` with its `bubbles` attribute initialized to `true` at the BluetoothGATTService representing the Service. Then remove this BluetoothGATTService from the Bluetooth tree.
    2. For each Service in *addedEntities*, add the BluetoothGATTService representing this Service to the Bluetooth tree. Then fire an event named `serviceadded` with its `bubbles` attribute initialized to `true` at the BluetoothGATTService.
    3. For each Service in *changedServices*, fire an event named `servicechanged` with its `bubbles` attribute initialized to `true` at the BluetoothGATTService representing the Service.

K.20. IDL EVENT HANDLERS

attribute EventHandler oncharacteristicvaluechanged

Event handler IDL attribute for the `characteristicvaluechanged` event type.

attribute EventHandler onserviceadded

Event handler IDL attribute for the `serviceadded` event type.

attribute EventHandler onservicechanged

Event handler IDL attribute for the `servicechanged` event type.

attribute EventHandler onserviceremoved

Event handler IDL attribute for the `serviceremoved` event type.

K.21. ERROR HANDLING

This section primarily defines the mapping from system errors to Javascript error names and allows UAs to retry certain operations. The retry logic and possible error distinctions are highly constrained by the operating system, so places these requirements don't reflect reality are likely spec bugs instead of browser bugs.

When a step in an algorithm that uses a Bluetooth GATT procedure ([[!BLUETOOTH41]] 3.G.4) says to "*Use the Bluetooth error recovery procedure*" with a promise, the UA MUST perform the following steps:

1. If the Bluetooth procedure completes with anything other than an code>Error Response ([[!BLUETOOTH41]] 3.F.3.4.1.1), the recovery procedure succeeds. Abort these steps.
2. If the procedure times out ([[!BLUETOOTH41]] 3.G.4.14) or the ATT Bearer ([[!BLUETOOTH41]] 3.G.2.4) is terminated for any reason, reject the promise with a NetworkError and abort these steps. The recovery procedure fails.
3. Take the following actions depending on the `Error Code`:

   `Invalid Handle`
   `Invalid PDU`
   `Invalid Offset`
   `Attribute Not Found`
   `Unsupported Group Type`

   These error codes indicate that something unexpected happened at the protocol layer, likely either due to a UA or device bug. Reject the promise with a NotSupportedError. The recovery procedure fails.

   `Invalid Attribute Value Length`

   Reject the promise with an InvalidModificationError. The recovery procedure fails.

   `Attribute Not Long`

   If this error code is received without having used a "Long" sub-procedure, this may indicate a device bug. Reject the promise with a NotSupportedError, and the recovery procedure fails.

   Otherwise, retry the GATT procedure without using a "Long" sub-procedure. If this is impossible due to the length of the value being written, reject the promise with an InvalidModificationError, and the recovery procedure fails.

   `Insufficient Authentication`
   `Insufficient Encryption`
   `Insufficient Encryption Key Size`

   The UA SHOULD attempt to increase the security level of the connection. If this attempt fails or the UA doesn't support any higher security, reject the promise with a SecurityError, and the recovery procedure fails. Otherwise, retry the GATT procedure at the new higher security level.

   `Insufficient Authorization`

   Reject the promise with a SecurityError, and the recovery procedure fails.

`Read Not Permitted`
`Write Not Permitted`
`Request Not Supported`
`Prepare Queue Full`
`Insufficient Resources`
`Unlikely Error`
Anything else

Reject the promise with a NotSupportedError. The recovery procedure fails.

K.22. UUIDS

A UUID string represents a 128-bit [[!RFC4122]] UUID. A *valid UUID* is a string that matches the [[!ECMAScript]] regexp `/^[0-9a-f]{8}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{4}-[0-9a-f]{12}$/`. That is, a valid UUID is lower-case and does not use the 16- or 32-bit abbreviations defined by the Bluetooth standard. All UUIDs returned from functions and attributes in this specification MUST be valid UUIDs. If a function in this specification takes a parameter whose type is UUID or a dictionary including a UUID attribute, and the argument passed in any UUID slot is not a valid UUID, the function MUST return a Promise rejected with a `TypeError` and abort its other steps.

This standard provides the `canonicalUUID()` function to map a 16- or 32-bit Bluetooth UUID alias to its 128-bit form.

Bluetooth devices are required to convert 16- and 32-bit UUIDs to 128-bit UUIDs before comparing them ([[BLUETOOTH41]] 3.F.3.2.1), but not all devices do so. To interoperate with these devices, if the UA has received a UUID from the device in one form (16-, 32-, or 128-bit), it should send other aliases of that UUID back to the device in the same form.

K.23. STANDARDIZED IDENTIFIERS

The Bluetooth standard defines numbers that identify services, characteristics, descriptors, and other entities. This section provides javascript names for these constants so they don't need to be replicated in each application.

readonly attribute BluetoothUuidsUnit unit

readonly attribute BluetoothUuidsService service

readonly attribute BluetoothUuidsCharacteristic characteristic

readonly attribute BluetoothUuidsDescriptor descriptor

UUID canonicalUUID(unsigned long alias)

Returns a 128-bit UUID given a 16- or 32-bit Bluetooth UUID alias. The algorithm for converting an alias to a full 128-bit UUID is defined in [[!BLUETOOTH41]] Volume 3 Part B Section 2.5.1: the top 32 bits of

"00000000-0000-1000-8000-00805f9b34fb" are replaced by the bits of the alias. For example, canonicalUUID(0xDEADBEEF) returns "deadbeef-0000-1000-8000-00805f9b34fb".

K.24. STANDARD GATT UNITS

Each standardized unit listed in [[!BLUETOOTH-NUMBERS-UNITS]] MUST be reflected into `navigator.bluetooth.uuids.unit` under the name listed under "Type" with `org.bluetooth.unit.` removed.

readonly attribute UUID unitless
`canonicalUUID(0x2700)`

readonly attribute UUID length.metre
`canonicalUUID(0x2701)`

readonly attribute UUID mass.kilogram
`canonicalUUID(0x2702)`

readonly attribute UUID time.second
`canonicalUUID(0x2703)`

readonly attribute UUID electric_current.ampere
`canonicalUUID(0x2704)`

readonly attribute UUID thermodynamic_temperature.kelvin
`canonicalUUID(0x2705)`

readonly attribute UUID amount_of_substance.mole
`canonicalUUID(0x2706)`

readonly attribute UUID luminous_intensity.candela
`canonicalUUID(0x2707)`

readonly attribute UUID area.square_metres
`canonicalUUID(0x2710)`

readonly attribute UUID volume.cubic_metres
`canonicalUUID(0x2711)`

readonly attribute UUID velocity.metres_per_second
`canonicalUUID(0x2712)`

readonly attribute UUID acceleration.metres_per_second_squared
`canonicalUUID(0x2713)`

readonly attribute UUID wavenumber.reciprocal_metre
`canonicalUUID(0x2714)`

readonly attribute UUID density.kilogram_per_cubic_metre
`canonicalUUID(0x2715)`

readonly attribute UUID surface_density.kilogram_per_square_metre
`canonicalUUID(0x2716)`

readonly attribute UUID specific_volume.cubic_metre_per_kilogram
`canonicalUUID(0x2717)`

readonly attribute UUID current_density.ampere_per_square_metre
`canonicalUUID(0x2718)`

readonly attribute UUID magnetic_field_strength.ampere_per_metre
`canonicalUUID(0x2719)`

readonly attribute UUID amount_concentration.mole_per_cubic_metre
`canonicalUUID(0x271A)`

readonly attribute UUID mass_concentration.kilogram_per_cubic_metre
`canonicalUUID(0x271B)`

readonly attribute UUID luminance.candela_per_square_metre
`canonicalUUID(0x271C)`

readonly attribute UUID refractive_index
`canonicalUUID(0x271D)`

readonly attribute UUID relative_permeability
`canonicalUUID(0x271E)`

readonly attribute UUID plane_angle.radian
`canonicalUUID(0x2720)`

readonly attribute UUID solid_angle.steradian
`canonicalUUID(0x2721)`

readonly attribute UUID frequency.hertz
`canonicalUUID(0x2722)`

readonly attribute UUID force.newton
`canonicalUUID(0x2723)`

readonly attribute UUID pressure.pascal
`canonicalUUID(0x2724)`

readonly attribute UUID energy.joule
`canonicalUUID(0x2725)`

readonly attribute UUID power.watt
`canonicalUUID(0x2726)`

readonly attribute UUID electric_charge.coulomb
`canonicalUUID(0x2727)`

readonly attribute UUID electric_potential_difference.volt
`canonicalUUID(0x2728)`

readonly attribute UUID capacitance.farad
`canonicalUUID(0x2729)`

readonly attribute UUID electric_resistance.ohm
`canonicalUUID(0x272A)`

readonly attribute UUID electric_conductance.siemens
`canonicalUUID(0x272B)`

readonly attribute UUID magnetic_flux.weber
`canonicalUUID(0x272C)`

readonly attribute UUID magnetic_flux_density.tesla
`canonicalUUID(0x272D)`

readonly attribute UUID inductance.henry
`canonicalUUID(0x272E)`

readonly attribute UUID thermodynamic_temperature.degree_celsius
`canonicalUUID(0x272F)`

readonly attribute UUID luminous_flux.lumen
`canonicalUUID(0x2730)`

readonly attribute UUID illuminance.lux
`canonicalUUID(0x2731)`

readonly attribute UUID activity_referred_to_a_radionuclide.becquerel
`canonicalUUID(0x2732)`

readonly attribute UUID absorbed_dose.gray
`canonicalUUID(0x2733)`

readonly attribute UUID dose_equivalent.sievert
`canonicalUUID(0x2734)`

readonly attribute UUID catalytic_activity.katal
`canonicalUUID(0x2735)`

readonly attribute UUID dynamic_viscosity.pascal_second
`canonicalUUID(0x2740)`

readonly attribute UUID moment_of_force.newton_metre
`canonicalUUID(0x2741)`

readonly attribute UUID surface_tension.newton_per_metre
`canonicalUUID(0x2742)`

readonly attribute UUID angular_velocity.radian_per_second
`canonicalUUID(0x2743)`

readonly attribute UUID angular_acceleration.radian_per_second_squared
`canonicalUUID(0x2744)`

readonly attribute UUID heat_flux_density.watt_per_square_metre
`canonicalUUID(0x2745)`

readonly attribute UUID heat_capacity.joule_per_kelvin
`canonicalUUID(0x2746)`

readonly attribute UUID specific_heat_capacity.joule_per_kilogram_kelvin
`canonicalUUID(0x2747)`

readonly attribute UUID specific_energy.joule_per_kilogram
`canonicalUUID(0x2748)`

readonly attribute UUID thermal_conductivity.watt_per_metre_kelvin
`canonicalUUID(0x2749)`

readonly attribute UUID energy_density.joule_per_cubic_metre
`canonicalUUID(0x274A)`

readonly attribute UUID electric_field_strength.volt_per_metre
`canonicalUUID(0x274B)`

readonly attribute UUID electric_charge_density.coulomb_per_cubic_metre
`canonicalUUID(0x274C)`

readonly attribute UUID surface_charge_density.coulomb_per_square_metre
`canonicalUUID(0x274D)`

readonly attribute UUID electric_flux_density.coulomb_per_square_metre
`canonicalUUID(0x274E)`

readonly attribute UUID permittivity.farad_per_metre
`canonicalUUID(0x274F)`

readonly attribute UUID permeability.henry_per_metre
`canonicalUUID(0x2750)`

readonly attribute UUID molar_energy.joule_per_mole
`canonicalUUID(0x2751)`

readonly attribute UUID molar_entropy.joule_per_mole_kelvin
`canonicalUUID(0x2752)`

readonly attribute UUID exposure.coulomb_per_kilogram
`canonicalUUID(0x2753)`

readonly attribute UUID absorbed_dose_rate.gray_per_second
`canonicalUUID(0x2754)`

readonly attribute UUID radiant_intensity.watt_per_steradian
`canonicalUUID(0x2755)`

readonly attribute UUID radiance.watt_per_square_metre_steradian
`canonicalUUID(0x2756)`

readonly attribute UUID catalytic_activity_concentration.katal_per_cubic_metre
`canonicalUUID(0x2757)`

readonly attribute UUID time.minute
`canonicalUUID(0x2760)`

readonly attribute UUID time.hour
`canonicalUUID(0x2761)`

readonly attribute UUID time.day
`canonicalUUID(0x2762)`

readonly attribute UUID plane_angle.degree
`canonicalUUID(0x2763)`

readonly attribute UUID plane_angle.minute
`canonicalUUID(0x2764)`

readonly attribute UUID plane_angle.second
`canonicalUUID(0x2765)`

readonly attribute UUID area.hectare
`canonicalUUID(0x2766)`

readonly attribute UUID volume.litre
`canonicalUUID(0x2767)`

readonly attribute UUID mass.tonne
`canonicalUUID(0x2768)`

readonly attribute UUID pressure.bar
`canonicalUUID(0x2780)`

readonly attribute UUID pressure.millimetre_of_mercury
`canonicalUUID(0x2781)`

readonly attribute UUID length.ångström
`canonicalUUID(0x2782)`

readonly attribute UUID length.nautical_mile
`canonicalUUID(0x2783)`

readonly attribute UUID area.barn
`canonicalUUID(0x2784)`

readonly attribute UUID velocity.knot
`canonicalUUID(0x2785)`

readonly attribute UUID logarithmic_radio_quantity.neper
`canonicalUUID(0x2786)`

readonly attribute UUID logarithmic_radio_quantity.bel
`canonicalUUID(0x2787)`

readonly attribute UUID length.yard
`canonicalUUID(0x27A0)`

readonly attribute UUID length.parsec
`canonicalUUID(0x27A1)`

readonly attribute UUID length.inch
`canonicalUUID(0x27A2)`

readonly attribute UUID length.foot
`canonicalUUID(0x27A3)`

readonly attribute UUID length.mile
`canonicalUUID(0x27A4)`

readonly attribute UUID pressure.pound_force_per_square_inch
`canonicalUUID(0x27A5)`

readonly attribute UUID velocity.kilometre_per_hour
`canonicalUUID(0x27A6)`

readonly attribute UUID velocity.mile_per_hour
`canonicalUUID(0x27A7)`

readonly attribute UUID angular_velocity.revolution_per_minute
`canonicalUUID(0x27A8)`

readonly attribute UUID energy.gram_calorie
`canonicalUUID(0x27A9)`

readonly attribute UUID energy.kilogram_calorie
`canonicalUUID(0x27AA)`

readonly attribute UUID energy.kilowatt_hour
`canonicalUUID(0x27AB)`

readonly attribute UUID thermodynamic_temperature.degree_fahrenheit
`canonicalUUID(0x27AC)`

readonly attribute UUID percentage
`canonicalUUID(0x27AD)`

readonly attribute UUID per_mille
`canonicalUUID(0x27AE)`

readonly attribute UUID period.beats_per_minute
`canonicalUUID(0x27AF)`

readonly attribute UUID electric_charge.ampere_hours
`canonicalUUID(0x27B0)`

readonly attribute UUID mass_density.milligram_per_decilitre
`canonicalUUID(0x27B1)`

readonly attribute UUID mass_density.millimole_per_litre
`canonicalUUID(0x27B2)`

readonly attribute UUID time.year
`canonicalUUID(0x27B3)`

readonly attribute UUID time.month
`canonicalUUID(0x27B4)`

readonly attribute UUID concentration.count_per_cubic_metre
`canonicalUUID(0x27B5)`

readonly attribute UUID irradiance.watt_per_square_metre
`canonicalUUID(0x27B6)`

readonly attribute UUID milliliter_per_kilogram_per_minute
`canonicalUUID(0x27B7)`

readonly attribute UUID mass.pound
`canonicalUUID(0x27B8)`

K.25. STANDARD GATT SERVICES

Each standardized service listed in [[!BLUETOOTH-NUMBERS-SERVICES]] MUST be reflected into `navigator.bluetooth.uuids.service` under the name listed under "SpecificationType" with `org.bluetooth.service.` removed.

readonly attribute UUID alert_notification
`canonicalUUID(0x1811)`

readonly attribute UUID battery_service
`canonicalUUID(0x180F)`

readonly attribute UUID blood_pressure
`canonicalUUID(0x1810)`

readonly attribute UUID current_time
`canonicalUUID(0x1805)`

readonly attribute UUID cycling_power
`canonicalUUID(0x1818)`

readonly attribute UUID cycling_speed_and_cadence
`canonicalUUID(0x1816)`

readonly attribute UUID device_information
`canonicalUUID(0x180A)`

readonly attribute UUID generic_access
`canonicalUUID(0x1800)`

readonly attribute UUID generic_attribute
`canonicalUUID(0x1801)`

readonly attribute UUID glucose
`canonicalUUID(0x1808)`

readonly attribute UUID health_thermometer
`canonicalUUID(0x1809)`

readonly attribute UUID heart_rate
`canonicalUUID(0x180D)`

readonly attribute UUID human_interface_device
`canonicalUUID(0x1812)`

readonly attribute UUID immediate_alert
`canonicalUUID(0x1802)`

readonly attribute UUID link_loss
`canonicalUUID(0x1803 )`

readonly attribute UUID location_and_navigation
`canonicalUUID(0x1819)`

readonly attribute UUID next_dst_change
`canonicalUUID(0x1807)`

readonly attribute UUID phone_alert_status
  `canonicalUUID(0x180E)`

readonly attribute UUID reference_time_update
  `canonicalUUID(0x1806)`

readonly attribute UUID running_speed_and_cadence
  `canonicalUUID(0x1814)`

readonly attribute UUID scan_parameters
  `canonicalUUID(0x1813)`

readonly attribute UUID tx_power
  `canonicalUUID(0x1804)`

readonly attribute UUID user_data
  `canonicalUUID(0x181C)`

The BluetoothServiceName enumeration allows users to pass the standardized services by name instead of looking up their UUIDs inside `navigator.bluetooth.uuids.service`. When used as a parameter to a function in this specification that accepts a BluetoothServiceUuid parameter, these enumeration values MUST be treated as equivalent to the UUID they refer to.

alert_notification
  refers to `canonicalUUID(0x1811)`

battery_service
  refers to `canonicalUUID(0x180F)`

blood_pressure
  refers to `canonicalUUID(0x1810)`

current_time
  refers to `canonicalUUID(0x1805)`

cycling_power
  refers to `canonicalUUID(0x1818)`

cycling_speed_and_cadence
  refers to `canonicalUUID(0x1816)`

device_information
  refers to `canonicalUUID(0x180A)`

generic_access
  refers to `canonicalUUID(0x1800)`

generic_attribute
  refers to `canonicalUUID(0x1801)`

glucose
refers to `canonicalUUID(0x1808)`

health_thermometer
refers to `canonicalUUID(0x1809)`

heart_rate
refers to `canonicalUUID(0x180D)`

human_interface_device
refers to `canonicalUUID(0x1812)`

immediate_alert
refers to `canonicalUUID(0x1802)`

link_loss
refers to `canonicalUUID(0x1803)`

location_and_navigation
refers to `canonicalUUID(0x1819)`

next_dst_change
refers to `canonicalUUID(0x1807)`

phone_alert_status
refers to `canonicalUUID(0x180E)`

reference_time_update
refers to `canonicalUUID(0x1806)`

running_speed_and_cadence
refers to `canonicalUUID(0x1814)`

scan_parameters
refers to `canonicalUUID(0x1813)`

tx_power
refers to `canonicalUUID(0x1804)`

user_data
refers to `canonicalUUID(0x181C)`

K.26. STANDARD GATT CHARACTERISTICS

Each standardized characteristic listed in [[!BLUETOOTH-NUMBERS-CHARACTERISTICS]] MUST be reflected into `navigator.bluetooth.uuids.characteristic` under the name listed under "SpecificationType" with `org.bluetooth.characteristic.` removed.

readonly attribute UUID aerobic_heart_rate_lower_limit
`canonicalUUID(0x2A7E)`

readonly attribute UUID aerobic_heart_rate_upper_limit
`canonicalUUID(0x2A84)`

readonly attribute UUID aerobic_threshold
`canonicalUUID(0x2A7F)`

readonly attribute UUID age
`canonicalUUID(0x2A80)`

readonly attribute UUID alert_category_id
`canonicalUUID(0x2A43)`

readonly attribute UUID alert_category_id_bit_mask
`canonicalUUID(0x2A42)`

readonly attribute UUID alert_level
`canonicalUUID(0x2A06)`

readonly attribute UUID alert_notification_control_point
`canonicalUUID(0x2A44)`

readonly attribute UUID alert_status
`canonicalUUID(0x2A3F)`

readonly attribute UUID anaerobic_heart_rate_lower_limit
`canonicalUUID(0x2A81)`

readonly attribute UUID anaerobic_heart_rate_upper_limit
`canonicalUUID(0x2A82)`

readonly attribute UUID anaerobic_threshold
`canonicalUUID(0x2A83)`

readonly attribute UUID gap.appearance
`canonicalUUID(0x2A01)`

readonly attribute UUID battery_level
`canonicalUUID(0x2A19)`

readonly attribute UUID blood_pressure_feature
`canonicalUUID(0x2A49)`

readonly attribute UUID blood_pressure_measurement
`canonicalUUID(0x2A35)`

readonly attribute UUID body_sensor_location
`canonicalUUID(0x2A38)`

readonly attribute UUID boot_keyboard_input_report
`canonicalUUID(0x2A22)`

readonly attribute UUID boot_keyboard_output_report
`canonicalUUID(0x2A32)`

readonly attribute UUID boot_mouse_input_report
`canonicalUUID(0x2A33)`

readonly attribute UUID csc_feature
`canonicalUUID(0x2A5C)`

readonly attribute UUID csc_measurement
`canonicalUUID(0x2A5B)`

readonly attribute UUID current_time
`canonicalUUID(0x2A2B)`

readonly attribute UUID cycling_power_control_point
`canonicalUUID(0x2A66)`

readonly attribute UUID cycling_power_feature
`canonicalUUID(0x2A65)`

readonly attribute UUID cycling_power_measurement
`canonicalUUID(0x2A63)`

readonly attribute UUID cycling_power_vector
`canonicalUUID(0x2A64)`

readonly attribute UUID database_change_increment
`canonicalUUID(0x2A99)`

readonly attribute UUID date_of_birth
`canonicalUUID(0x2A85)`

readonly attribute UUID date_of_threshold_assessment
`canonicalUUID(0x2A86)`

readonly attribute UUID date_time
`canonicalUUID(0x2A08)`

readonly attribute UUID day_date_time
`canonicalUUID(0x2A0A)`

readonly attribute UUID day_of_week
`canonicalUUID(0x2A09)`

readonly attribute UUID gap.device_name
`canonicalUUID(0x2A00)`

readonly attribute UUID dst_offset
`canonicalUUID(0x2A0D)`

readonly attribute UUID email_address
`canonicalUUID(0x2A87)`

readonly attribute UUID exact_time_256
`canonicalUUID(0x2A0C)`

readonly attribute UUID fat_burn_heart_rate_lower_limit
`canonicalUUID(0x2A88)`

readonly attribute UUID fat_burn_heart_rate_upper_limit
`canonicalUUID(0x2A89)`

readonly attribute UUID firmware_revision_string
`canonicalUUID(0x2A26)`

readonly attribute UUID first_name
`canonicalUUID(0x2A8A)`

readonly attribute UUID five_zone_heart_rate_limits
`canonicalUUID(0x2A8B)`

readonly attribute UUID gender
`canonicalUUID(0x2A8C)`

readonly attribute UUID glucose_feature
`canonicalUUID(0x2A51)`

readonly attribute UUID glucose_measurement
`canonicalUUID(0x2A18)`

readonly attribute UUID glucose_measurement_context
`canonicalUUID(0x2A34)`

readonly attribute UUID hardware_revision_string
`canonicalUUID(0x2A27)`

readonly attribute UUID heart_rate_control_point
`canonicalUUID(0x2A39)`

readonly attribute UUID heart_rate_max
`canonicalUUID(0x2A8D)`

readonly attribute UUID heart_rate_measurement
`canonicalUUID(0x2A37)`

readonly attribute UUID height
`canonicalUUID(0x2A8E)`

readonly attribute UUID hid_control_point
`canonicalUUID(0x2A4C)`

readonly attribute UUID hid_information
`canonicalUUID(0x2A4A)`

readonly attribute UUID hip_circumference
`canonicalUUID(0x2A8F)`

readonly attribute UUID ieee_11073-20601_regulatory_certification_data_list
`canonicalUUID(0x2A2A)`

readonly attribute UUID intermediate_blood_pressure
`canonicalUUID(0x2A36)`

readonly attribute UUID intermediate_temperature
`canonicalUUID(0x2A1E)`

readonly attribute UUID language
`canonicalUUID(0x2AA2)`

readonly attribute UUID last_name
`canonicalUUID(0x2A90)`

readonly attribute UUID ln_control_point
`canonicalUUID(0x2A6B)`

readonly attribute UUID ln_feature
`canonicalUUID(0x2A6A)`

readonly attribute UUID local_time_information
`canonicalUUID(0x2A0F)`

readonly attribute UUID location_and_speed
`canonicalUUID(0x2A67)`

readonly attribute UUID manufacturer_name_string
`canonicalUUID(0x2A29)`

readonly attribute UUID maximum_recommended_heart_rate
`canonicalUUID(0x2A91)`

readonly attribute UUID measurement_interval
`canonicalUUID(0x2A21)`

readonly attribute UUID model_number_string
`canonicalUUID(0x2A24)`

readonly attribute UUID navigation
`canonicalUUID(0x2A68)`

readonly attribute UUID new_alert
`canonicalUUID(0x2A46)`

readonly attribute UUID gap.peripheral_preferred_connection_parameters
`canonicalUUID(0x2A04)`

readonly attribute UUID gap.peripheral_privacy_flag
`canonicalUUID(0x2A02)`

readonly attribute UUID pnp_id
`canonicalUUID(0x2A50)`

readonly attribute UUID position_quality
`canonicalUUID(0x2A69)`

readonly attribute UUID protocol_mode
`canonicalUUID(0x2A4E)`

readonly attribute UUID gap.reconnection_address
`canonicalUUID(0x2A03)`

readonly attribute UUID record_access_control_point
`canonicalUUID(0x2A52)`

readonly attribute UUID reference_time_information
`canonicalUUID(0x2A14)`

readonly attribute UUID report
`canonicalUUID(0x2A4D)`

readonly attribute UUID report_map
`canonicalUUID(0x2A4B)`

readonly attribute UUID resting_heart_rate
`canonicalUUID(0x2A92)`

readonly attribute UUID ringer_control_point
`canonicalUUID(0x2A40)`

readonly attribute UUID ringer_setting
`canonicalUUID(0x2A41)`

readonly attribute UUID rsc_feature
`canonicalUUID(0x2A54)`

readonly attribute UUID rsc_measurement
`canonicalUUID(0x2A53)`

readonly attribute UUID sc_control_point
`canonicalUUID(0x2A55)`

readonly attribute UUID scan_interval_window
`canonicalUUID(0x2A4F)`

readonly attribute UUID scan_refresh
`canonicalUUID(0x2A31)`

readonly attribute UUID sensor_location
`canonicalUUID(0x2A5D)`

readonly attribute UUID serial_number_string
`canonicalUUID(0x2A25)`

readonly attribute UUID gatt.service_changed
`canonicalUUID(0x2A05)`

readonly attribute UUID software_revision_string
`canonicalUUID(0x2A28)`

readonly attribute UUID sport_type_for_aerobic_and_anaerobic_thresholds
`canonicalUUID(0x2A93)`

readonly attribute UUID supported_new_alert_category
`canonicalUUID(0x2A47)`

readonly attribute UUID supported_unread_alert_category
`canonicalUUID(0x2A48)`

readonly attribute UUID system_id
`canonicalUUID(0x2A23)`

readonly attribute UUID temperature_measurement
`canonicalUUID(0x2A1C)`

readonly attribute UUID temperature_type
`canonicalUUID(0x2A1D)`

readonly attribute UUID three_zone_heart_rate_limits
`canonicalUUID(0x2A94)`

readonly attribute UUID time_accuracy
`canonicalUUID(0x2A12)`

readonly attribute UUID time_source
`canonicalUUID(0x2A13)`

The BluetoothCharacteristicName enumeration allows users to pass the standardized characteristics by name instead of looking up their UUIDs inside `navigator.bluetooth.uuids.characteristic`. When used as a parameter to a function in this specification that accepts a BluetoothCharacteristicUuid parameter, these enumeration values MUST be treated as equivalent to the UUID they refer to.

aerobic_heart_rate_lower_limit
refers to `canonicalUUID(0x2A7E)`

aerobic_heart_rate_upper_limit
refers to `canonicalUUID(0x2A84)`

aerobic_threshold
refers to `canonicalUUID(0x2A7F)`

age
refers to `canonicalUUID(0x2A80)`

alert_category_id
refers to `canonicalUUID(0x2A43)`

alert_category_id_bit_mask
refers to `canonicalUUID(0x2A42)`

alert_level
refers to `canonicalUUID(0x2A06)`

alert_notification_control_point
refers to `canonicalUUID(0x2A44)`

alert_status
refers to `canonicalUUID(0x2A3F)`

anaerobic_heart_rate_lower_limit
refers to `canonicalUUID(0x2A81)`

anaerobic_heart_rate_upper_limit
refers to `canonicalUUID(0x2A82)`

anaerobic_threshold
refers to `canonicalUUID(0x2A83)`

gap.appearance
refers to `canonicalUUID(0x2A01)`

battery_level
refers to `canonicalUUID(0x2A19)`

blood_pressure_feature
refers to `canonicalUUID(0x2A49)`

blood_pressure_measurement
refers to `canonicalUUID(0x2A35)`

body_sensor_location
refers to `canonicalUUID(0x2A38)`

boot_keyboard_input_report
refers to `canonicalUUID(0x2A22)`

boot_keyboard_output_report
refers to `canonicalUUID(0x2A32)`

boot_mouse_input_report
refers to `canonicalUUID(0x2A33)`

csc_feature
refers to `canonicalUUID(0x2A5C)`

csc_measurement
refers to `canonicalUUID(0x2A5B)`

current_time
refers to `canonicalUUID(0x2A2B)`

cycling_power_control_point
refers to `canonicalUUID(0x2A66)`

cycling_power_feature
refers to `canonicalUUID(0x2A65)`

cycling_power_measurement
refers to `canonicalUUID(0x2A63)`

cycling_power_vector
refers to `canonicalUUID(0x2A64)`

database_change_increment
refers to `canonicalUUID(0x2A99)`

date_of_birth
refers to `canonicalUUID(0x2A85)`

date_of_threshold_assessment
refers to `canonicalUUID(0x2A86)`

date_time
refers to `canonicalUUID(0x2A08)`

day_date_time
refers to `canonicalUUID(0x2A0A)`

day_of_week
refers to `canonicalUUID(0x2A09)`

gap.device_name
refers to `canonicalUUID(0x2A00)`

dst_offset
refers to `canonicalUUID(0x2A0D)`

email_address
refers to `canonicalUUID(0x2A87)`

exact_time_256
refers to `canonicalUUID(0x2A0C)`

fat_burn_heart_rate_lower_limit
refers to `canonicalUUID(0x2A88)`

fat_burn_heart_rate_upper_limit
refers to `canonicalUUID(0x2A89)`

firmware_revision_string
refers to `canonicalUUID(0x2A26)`

first_name
refers to `canonicalUUID(0x2A8A)`

five_zone_heart_rate_limits
refers to `canonicalUUID(0x2A8B)`

gender
refers to `canonicalUUID(0x2A8C)`

glucose_feature
refers to `canonicalUUID(0x2A51)`

glucose_measurement
refers to `canonicalUUID(0x2A18)`

glucose_measurement_context
refers to `canonicalUUID(0x2A34)`

hardware_revision_string
refers to `canonicalUUID(0x2A27)`

heart_rate_control_point
refers to `canonicalUUID(0x2A39)`

heart_rate_max
refers to `canonicalUUID(0x2A8D)`

heart_rate_measurement
refers to `canonicalUUID(0x2A37)`

height
refers to `canonicalUUID(0x2A8E)`

hid_control_point
refers to `canonicalUUID(0x2A4C)`

hid_information
refers to `canonicalUUID(0x2A4A)`

hip_circumference
refers to `canonicalUUID(0x2A8F)`

ieee_11073-20601_regulatory_certification_data_list
refers to `canonicalUUID(0x2A2A)`

intermediate_blood_pressure
refers to `canonicalUUID(0x2A36)`

intermediate_temperature
refers to `canonicalUUID(0x2A1E)`

language
refers to `canonicalUUID(0x2AA2)`

last_name
refers to `canonicalUUID(0x2A90)`

ln_control_point
refers to `canonicalUUID(0x2A6B)`

ln_feature
refers to `canonicalUUID(0x2A6A)`

local_time_information
refers to `canonicalUUID(0x2A0F)`

location_and_speed
refers to `canonicalUUID(0x2A67)`

manufacturer_name_string
refers to `canonicalUUID(0x2A29)`

maximum_recommended_heart_rate
refers to `canonicalUUID(0x2A91)`

measurement_interval
refers to `canonicalUUID(0x2A21)`

model_number_string
refers to `canonicalUUID(0x2A24)`

navigation
refers to `canonicalUUID(0x2A68)`

new_alert
refers to `canonicalUUID(0x2A46)`

gap.peripheral_preferred_connection_parameters
refers to `canonicalUUID(0x2A04)`

gap.peripheral_privacy_flag
refers to `canonicalUUID(0x2A02)`

pnp_id
refers to `canonicalUUID(0x2A50)`

position_quality
refers to `canonicalUUID(0x2A69)`

protocol_mode
refers to `canonicalUUID(0x2A4E)`

gap.reconnection_address
refers to `canonicalUUID(0x2A03)`

record_access_control_point
refers to `canonicalUUID(0x2A52)`

reference_time_information
refers to `canonicalUUID(0x2A14)`

report
refers to `canonicalUUID(0x2A4D)`

report_map
refers to `canonicalUUID(0x2A4B)`

resting_heart_rate
refers to `canonicalUUID(0x2A92)`

ringer_control_point
refers to `canonicalUUID(0x2A40)`

ringer_setting
refers to `canonicalUUID(0x2A41)`

rsc_feature
refers to `canonicalUUID(0x2A54)`

rsc_measurement
refers to `canonicalUUID(0x2A53)`

sc_control_point
refers to `canonicalUUID(0x2A55)`

scan_interval_window
refers to `canonicalUUID(0x2A4F)`

scan_refresh
refers to `canonicalUUID(0x2A31)`

sensor_location
refers to `canonicalUUID(0x2A5D)`

serial_number_string
refers to `canonicalUUID(0x2A25)`

gatt.service_changed
refers to `canonicalUUID(0x2A05)`

software_revision_string
refers to `canonicalUUID(0x2A28)`

sport_type_for_aerobic_and_anaerobic_thresholds
refers to `canonicalUUID(0x2A93)`

supported_new_alert_category
refers to `canonicalUUID(0x2A47)`

supported_unread_alert_category
refers to `canonicalUUID(0x2A48)`

system_id
refers to `canonicalUUID(0x2A23)`

temperature_measurement
refers to `canonicalUUID(0x2A1C)`

temperature_type
refers to `canonicalUUID(0x2A1D)`

three_zone_heart_rate_limits
refers to `canonicalUUID(0x2A94)`

time_accuracy
refers to `canonicalUUID(0x2A12)`

time_source
refers to `canonicalUUID(0x2A13)`

K.27. STANDARD GATT DESCRIPTORS

Each standardized descriptor listed in [[!BLUETOOTH-NUMBERS-DESCRIPTORS]] MUST be reflected into `navigator.bluetooth.uuids.descriptor` under the name listed under "SpecificationType" with `org.bluetooth.descriptor.` removed.

readonly attribute UUID gatt.characteristic_extended_properties
`canonicalUUID(0x2900)`

readonly attribute UUID gatt.characteristic_user_description
`canonicalUUID(0x2901)`

readonly attribute UUID gatt.client_characteristic_configuration
`canonicalUUID(0x2902)`

readonly attribute UUID gatt.server_characteristic_configuration
`canonicalUUID(0x2903)`

readonly attribute UUID gatt.characteristic_presentation_format
`canonicalUUID(0x2904)`

readonly attribute UUID gatt.characteristic_aggregate_format
`canonicalUUID(0x2905)`

readonly attribute UUID valid_range
`canonicalUUID(0x2906)`

readonly attribute UUID external_report_reference
`canonicalUUID(0x2907)`

readonly attribute UUID report_reference
`canonicalUUID(0x2908)`

The BluetoothDescriptorName enumeration allows users to pass the standardized descriptors by name instead of looking up their UUIDs inside `navigator.bluetooth.uuids.descriptor`. When used as a parameter to a function in this specification that accepts a BluetoothDescriptorUuid parameter, these enumeration values MUST be treated as equivalent to the UUID they refer to.

gatt.characteristic_extended_properties
refers to `canonicalUUID(0x2900)`

gatt.characteristic_user_description
refers to `canonicalUUID(0x2901)`

gatt.client_characteristic_configuration
refers to `canonicalUUID(0x2902)`

gatt.server_characteristic_configuration
refers to `canonicalUUID(0x2903)`

gatt.characteristic_presentation_format
refers to `canonicalUUID(0x2904)`

gatt.characteristic_aggregate_format
refers to `canonicalUUID(0x2905)`

valid_range
refers to `canonicalUUID(0x2906)`

external_report_reference
refers to `canonicalUUID(0x2907)`

report_reference
refers to `canonicalUUID(0x2908)`

K.28. INTERFACE WIRING

readonly attribute Bluetooth bluetooth
Provides access to Bluetooth APIs.

L. SECURE ELEMENT API

Unofficial Draft 08 October 2014

Latest editor's draft:
http://opoto.github.io/secure-element

Editors:
Olivier Potonniée, Gemalto
Ming Yin, Deutsche Telekom

A secure element is a tamper proof device, providing a secure storage and execution environment for sensitive data and processing. It offers both physical and logical protection against attacks, ensuring integrity and confidentiality of its content.

This specification defines a communication interface between a web application and a secure element. It makes no assumption on the secure element type, application domain, or physical communication media.

This work-in-progress specification is an unofficial draft. Developers are strongly discouraged to implement it at this stage. If you are interested in implementing or using this API, we encourage you to contact the editors or post on the working group mailing list.

See Changes section for history details.

L.1. INTRODUCTION

The *Secure Element API* defined herein allows applications to interact with secure elements. Considered secure elements are those complying to [[!ISO7816-4]], which defines a command/response protocol, based on structured APDU (Application Data Unit).

L.2. TECHNICAL BACKGROUND

Secure elements addressed by this specification are micro controllers that may come in different form factors, such as:

- Smart cards. The chip is embedded in a plastic card usually of the size of a typical credit card. The card may show physical contacts to communicate with the chip, or the chip may support NFC (Near Field Communication), in which case the plastic card embeds an antenna. Some cards also support both communication methods.
- UICC (Universal Integrated Circuit Card) are smart cards used in cellular telephony, which may be delivered in different sizes. They are often called SIM, which is actually the name of the application hosted by the UICC to access GSM networks. UICC may however host other applications.
- Smart SD cards have a similar form as usual SD cards, but internally include a secure element, and support an extended set of SD commands to communicate with the secure element. Some of these smart SD cards also support NFC.
- Embedded secure elements, which are chips directly bonded on the device mother board. Unlike other form factors, this one does not allow interchanging or extracting the secure element, it is permanently attached to the device.

Similarly to a computer, a secure element may host one or multiple applications. Typical applications are mobile network authentication (SIM cards), payment (credit cards), authentication and signature (corporate badges, eID, etc.), loyalty, ticketing (public transports). But these are only examples, many other applications have been and can be deployed.

Applications hosted by the secure elements are commonly named *on-card* applications. Considering the limited, if any, user interface of these devices, and application can only be useful for a user if there is also an off-card application part, which handle the dialog with the user, or with external computing resources. Examples of *off-card* applications are ATM for payment, mail applications for signature, access control doors for authentication, etc. This specification defines the API to be used by off-card applications based on web technologies.

L.3. USE CASES

This specification allows development of web applications making use of these secure element applications. Some typical use cases that applications can address based on this API include:

- **Authentication:** Instead of user name and password, access to an online service may be protected by a strong authentication mechanism, based on credentials stored and processed in a secure element. In web-based operating systems, system applications such as VPN (Virtual Private Network) or eMail application may use of the secure element to authenticate the user.
- **Digital Signature:** Applications may use the secure element to digitally sign a document or any data with a key stored in this secure element. The signature operation itself is executed inside the secure element, ensuring both the integrity of the signature and the confidentiality of the key used in this process. For instance, this could be used by an eMail application to sign emails sent by the user. Or by a government web application to sign a online administrative request.
- **Payment:** Online commerce may use widely used smart credit cards, or specific payment applications, to enforce the security of online transactions. On cellular telephony environment, the on-card payment application may be hosted on the SIM card, alleviating the need for the user to handle multiple physical devices.
- **Credential provisioning:** The content of a secure element may be updated to install, update or remove an application or any credential it may host. For instance a public transport application may offer a user to credit her NFC-enabled transport card with tickets bought online. Or a corporate intranet web application may offer employees to renew online the X.509 certificates hosted in their corporate badge, so that they can do this operation from anywhere just before these certificates expire.

Whatever the form factor listed above, secure element considered in this specification implement the same [[!ISO7816-4]] transport protocol. The physical media (USB, NFC, or any other wired or wireless technique) used in this communication is abstracted by the API defined in this specification.

L.4. RELATIONSHIP TO OTHER W3C APIS

This specification, although addressing some concepts similar to other W3C specifications, has distinct use cases and offer different level of services:

- The current NFC API draft specification [[NFC]] defines an API allowing to exchange NDEF messages with NFC tags or peers. While the Secure Element API specified herein allows web applications to send commands to secure elements wired or plugged in the device, or wirelessly connected to the device thanks to NFC technology. The difference between the different communication links (wired or NFC) is only visible through secure element type in this API, but does not impact the way applications would interact with the secure element. As such there is no overlap of functionalities between the two APIs.
- The Web Cryptography API draft specification [[WEBCRYPTO]] defines an API allowing a web application to invoke cryptographic services. Its implementation is independent from the underlying layers performing the actual cryptographic operations: it might be pure software, or use a dedicated hardware such as a secure element or a TPM. As such there is not overlap between the two APIs. Nevertheless one can imagine that the User Agent implementing the Web Cryptography API may rely on the Secure Element API, but this will be implementation dependent.

This specification defines conformance criteria that apply to a single product: the *user agent* that implements the interfaces that it contains.

Implementations that use ECMAScript to implement the APIs defined in this specification MUST implement them in a manner consistent with the ECMAScript Bindings defined in the Web IDL specification [[!WEBIDL]], as this specification uses that specification and terminology.

L.5. DEPENDENCIES

This specification depends on interfaces and concepts defined in the following specifications.

[[!HTML]]: *event handler*.

[[!DOM4]]: the *`Event`* and *`DOMException`* interfaces, the concept of *firing an event*.

[[!ES6]]: the *`Promise`* object type.

[[!RFC6454]]: The concepts of *origin* and *same-origin*, as well as the algorithm for serializing an origin.

[[!GP-AC]]: The *Access Rules* and *Access Control Enforcer* concepts.

Some secure element concepts are defined in ISO/IEC specifications:

- *ATR* (Answer to Reset) is defined in [[!ISO7816-3]]
- *APDU command*, *APDU response*, *class byte*, *instruction byte*, *parameter bytes*, *data field bytes*, *status word*, *basic channel* and *logical channel* are defined in [[!ISO7816-4]]
- *AID* (Application Identifier) is defined in [[!ISO7816-5]].
- *UICC* is defined in [[!ETSI-102216]]

L.6. SECURITY AND PRIVACY CONSIDERATIONS

Using a secure element may bring additional security to a web application, but is not sufficient to ensure the application is secure. In particular, developers using the Secure Element API should be aware of the following security considerations:

- **Communication** between the secure element and the web application has to be secured, in order to ensure the confidentiality and integrity of the message exchange. This can either be achieved if the web application using the secure element API executes in a trusted execution environment offering guarantees on the integrity and confidentiality of the communication link. Or it can be provided programmatically by encryption and/or MACing of the messages exchanged with the secure element (e.g. using GlobalPlatform's secure messaging technology [[GP]]). The off-card processing of these messages has then to be done in a trusted execution environment, which may be on the device to which the secure element is connected, or on a remote device (e.g. the web application's originating server).

- **Interface** between the user and the web application typically consists in displayed text and images (e.g. a transaction confirmation dialog), and user inputs (e.g. a PIN code). Protecting this interface is out of the scope of this specification. If the application requires such guarantee, it should restrict its execution on Trusted Execution Environments.
- **Access** to secure element applications should be restricted to authorized parties. Secure element embedded application usually enforce such control by requiring authentication of the off-card communicating party, for instance by asking user to present a PIN to unlock access, or performing a mutual authentication before any sensitive operation.
  There is however a risk of denial of service (DoS) attacks, where an attacker would e.g. deliberately present multiple invalid PIN values to block the secure element, or sending burst of commands preventing legitimate applications to execute optimally. In order to mitigate this risk, this specification requires the web application runtime to implement the access control mechanism defined in Access Control section.
- **Traceability** of the user may be facilitated by the unique identifying information the secure element may contain. Here again, the access control defined in Access Control section will ensure only trusted applications have access to the secure element API.

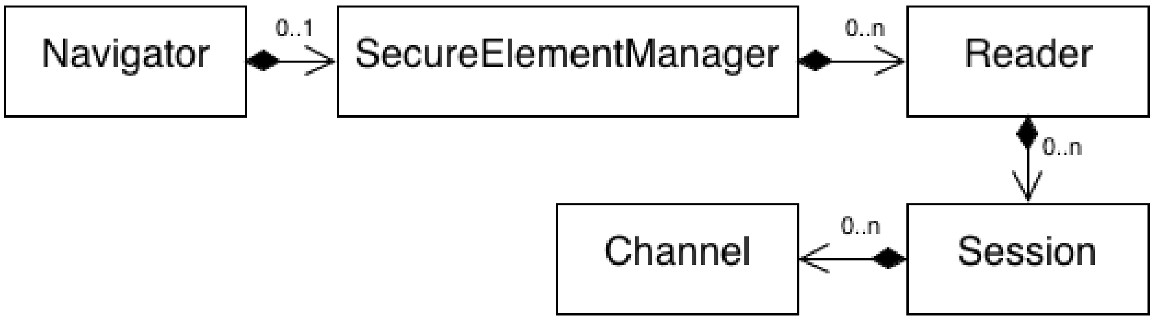
L.7. SECURE ELEMENT SERVICES

Communication with a secure element is performed through a *reader*, which unlike its name suggests it not only able to read content from secure element, but also to write or send any application specific command. Given the removal nature of many secure elements, a reader may be empty, meaning that no secure element is connected to this reader. For instance, a USB smart card reader may be connected to a computer, but no smart card is inserted. Or a device may support NFC interactions with secure elements, but none is in the field. For this reason, the API provides a mean to query the list of available readers, and for each of them if they are empty or if a secure element is present. In addition, this specification defines events that are triggered when a secure element is connected to a reader, and also when it is disconnected.

Once a web application is informed that a reader has a connected secure element, it can open a *session* with it. Opening a session establishes the communication between the web application and the secure element, and provides a session object to the application. However a secure element is just an application container, and the web application still needs to identify the application with which it wants to communicate.

To this intent, the session object provides a mean to open a *channel* with a specific secure element application, which is uniquely identified by an AID. The web application runtime and the secure element may then perform some internal security checking to ensure the web application is allowed to connect to this secure element application. If authorized, the returned channel object is the one providing the method to send commands to the secure element application, and get the corresponding responses.

The above steps required to send a command to a secure element creates a set of chained objects. A reader may have several opened sessions, which may have several opened channels. Each object has a `close()` method to release all ressources associated to the target object, and invoke `close()` method on all underlying objects in the chain.



The figure above represents the class relationships between the secure element entities introduced in this specification.

L.8. ACCESS CONTROL

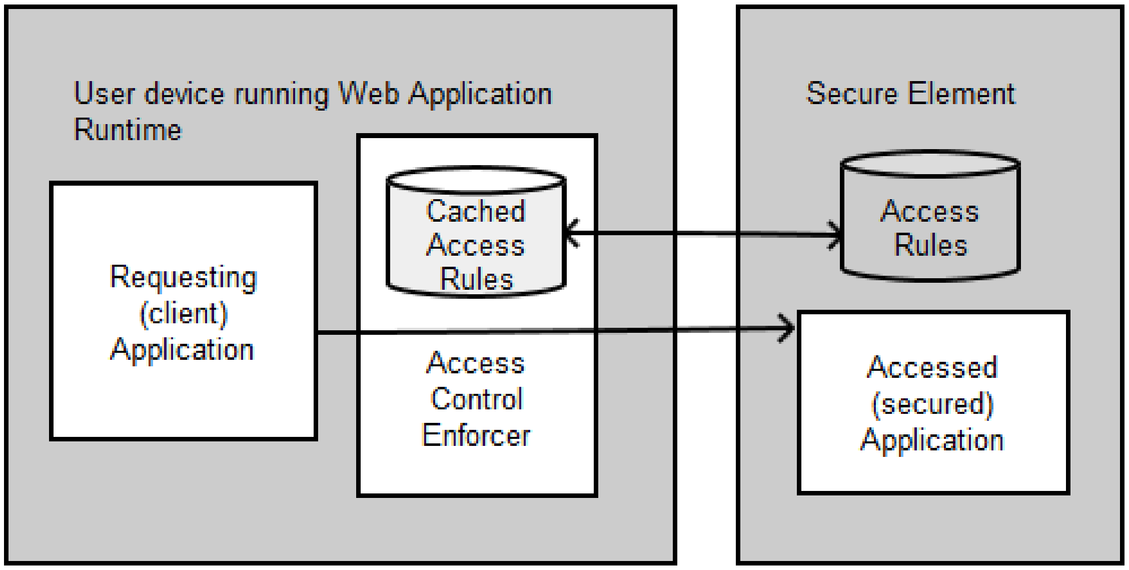
In order to make sure only trusted applications are allowed to use this API, the web application runtime MUST implement the access control defined in [[GP-AC]], which defines a simple mechanism that protects legitimate users using non-compromised devices from malicious applications. Note that it does not protect from a compromised device that would not properly implement the Access Control Enforcer, which is addressed by the internal protection of the secure element itself (using e.g. PIN or secure messaging)

L.8.1. Overall architecture

To control which applications running on a user device are allowed to access secure element applications, several entities are involved:

- The secure element hosts a list of Access Rules. An access rule contains the AID of the secure element application to control access to, and the identifier of the requesting application running on the device, as well as a filter on authorized APDUs, or a simple boolean to authorize all or no communications.
- An Access Control Enforcer is running on the device of the client application, inside the web application runtime. Any attempt to establish a communication with a secure element application from this device

triggers this enforcer, which queries the secure element to get the access rules (and usually cache them), and check that the requesting application is authorized to communicate with the targeted secure element application.


User device running Web Application Runtime
Requesting (client) Application
Cached Access Rules
Access Control Enforcer
Secure Element
Access Rules
Accessed (secured) Application

The figure above shows the overall access control architecture.

Sections below describe how the Global Platform Access Control is applied to web applications. Details of the Global Platform Access Control mechanisms itself are defined in [[!GP-AC]].

L.8.2. Trusted application identifier

The Access Control Enforcer uses an application identifier to check whether the application is white listed in Access Rules. This identifier needs to be trustworthy so that only authorized application may have a given identifier. The web application runtime computes the application identifier using the following algorithm:

- let `origin` be the ASCII serialization of the web application origin as defined in [[!RFC6454]]
- set the application identifier to be the SHA-1 digest of this `origin` value. The SHA-1 hash function is used here because the GlobalPlatform Access Control specification for now only supports 20 bytes long identifiers. A stronger algorithm will be used as soon as GlobalPlatform updates its specification to support longer values.

All applications that are same-origin will get the same application identifier, hence will be granted the same access rules.

While applying the GlobalPlatform Access Control process, if the Access Control Enforcer detects the Secure Element does not implement the GlobalPlatform Access Control (it does not have access rules), the Access Control Enforcer may use its own policy to grant or refuse access to the Secure Element API.

L.8.3. Additional security rules

To be eligible to gain access to the Secure Element API, a web application must meet the following requirements:

- The web application MUST be fetched using HTTPS protocol.
- The TLS/SSL server certificate of this HTTPS connection MUST be trusted and valid:
    - Its subject MUST match the hosting domain name.
    - Its validity dates MUST include the current date provided by the execution runtime.
    - The issuance signature chain MUST be valid.
    - The issuance chain MUST be rooted by a CA trusted by the web application runtime.
- The user MUST NOT be able to bypass these rules. Some browsers offer users to define "exceptions" to allow connections to a HTTPS URL even if SSL server certificate is invalid. Such exception MUST NOT be used to allow access to the Secure Element API.

L.9. NAVIGATOR INTERFACE

The Navigator exposes the secure element service.

readonly attribute SecureElementManager? secureElementManager
When getting the *secureElementManager* attribute, the user agent MUST return the SecureElementManager object that provides access to available secure element readers. If the user agent doesn't support secure element features, it MUST then return `undefined`.

L.10. SECUREELEMENTMANAGER INTERFACE

The SecureElementManager interface provides access to secure element readers, and is the source of events notifying the presence of secure elements.

readonly attribute Reader[] readers
This attribute MUST return the list of available readers in which a secure element may be present. Its value MUST be an empty array if no reader is available. It MUST be `null` if the `close()` method has been closed on this SecureElementManager object. Several requests of this attribute MAY return a different array value, because new readers may become available, while others may be disconnected.

attribute EventHandler? onsepresent
Event handler for the SE-present event. This event MUST be triggered each time any of the following situations occurs:

- Application starts while a secure element is present in a reader
- Application is running, a reader which was already listed in the readers attribute but had no secure element now detects a present secure element.

- Application is running, a new reader is detected and has a secure element present.

attribute EventHandler? onseremoval

Event handler for the SE-removal event. This event MUST be triggered when a secure element which was present in a reader is not present anymore (it has been unplugged, or is out of reach if it was connected through wireless communication). As soon as this event is triggered, all Reader, Session and Channel objects providing access to this secure element are marked as closed. Calling any method other than `close()` on associated readers, sessions, or channels MUST fail with an SEClosedException error.

Promise<void> close()

This method closes all readers and descendent Session and Channel objects. When invoked, the user agent MUST run the following steps:

1. Let *promise* be a newly-created Promise object.
2. Return *promise* and continue the following steps asynchronously.
3. If the `close()` method has already been called on this object, then resolve *promise*.
4. Let *countdown* be the number of readers in the `readers` attribute, and *error* an object initially `undefined`.
5. Invoke `close()` method on each Reader object in the array returned by the `readers` attribute.
6. Let *readerpromises* be the set of Promise objects returned by these `close()` invocations.
7. Set the `readers` attribute value to `null`.
8. If *countdown* is 0, then resolve *promise* with this SecureElementManager object.
9. When a *readerpromises* element is fulfilled, *countdown* is decremented. If *countdown* is 0 and *error* is `undefined`, then resolve *promise*. If *countdown* is 0 and *error* is not `undefined`, then reject *promise* with the *error* value.
10. When a *readerpromises* element is rejected, *countdown* is decremented. If *error* is `undefined` then set it to the rejected value. If *countdown* is 0 then reject *promise* with *error* value.

L.11. READER INTERFACE

Readers connected to this device are accessible through the Reader interface. A reader is the connector to a secure element. Given the removable nature of some secure elements, a reader may or may not have a secure element present. A reader may have at most one present secure element simultaneously. A reader MAY for instance be a UICC slot, a USB smart card reader, an NFC interface, or a mother board slot where a embedded secure element is wired.

readonly attribute boolean isSEPresent
This attribute MUST return true if a secure element is present in this reader. It MUST return false otherwise.

readonly attribute DOMString name
This attribute MUST return the name of the reader. This is an arbitrary name set by the system. It MAY be computed based on reader provided data.

readonly attribute SecureElementType secureElementType
This attribute MUST return the SecureElementType value best matching the type of the secure element this reader gives access to. This information may be useful for applications that target a specific secure element type. It may also be used to build the application user interface, to represent the secure element in a realistic way.

readonly attribute ConnectivityType connectivityType
This attribute MUST return the ConnectivityType value matching the connectivity used by the reader to communicate with the secure element.

Promise<Session> openSession()
This method establishes a communication link with a secure element. There may be several sessions opened at the same time, hence a session MUST NOT lock access to the secure element.

Promise<void> close()
This method closes all sessions opened by this reader, and their descendent Channel objects. Invoking `close()` method on an already closed reader is an idempotent operation.

L.12. SECUREELEMENTTYPE ENUM

The SecureElementType enum identifies the type of the secure element a reader gives access to.

uicc
The secure element is a UICC used by the device to connect to a mobile network.

smartcard
The secure element is a smart card.

chip
The secure element is a dedicated chip in the device.

sd
The secure element is a SD card, and may be unplugged.

other
For any other secure element type not listed above.

L.13. CONNECTIVITYTYPE ENUM

The ConnectivityType enum identifies the type of the secure element a reader gives access to.

embedded
The secure element is physically attached to the device, and cannot be removed, at least without powering off the device.

plugged
The secure element is plugged to the device, and can be unplugged.

wireless
The secure element is accessed though wireless communication, such as NFC. It can be disconnected.

L.14. SESSION INTERFACE

A Session represent a connection session to one of the Secure Elements available on the device. These objects can be used to get a communication channel with an application hosted by the Secure Element.

readonly attribute Reader reader
This attribute MUST return the reader object from which this session object was created.

readonly attribute Uint8Array? atr
This attribute MUST return the Answer to Reset provided by the secure element, or `null` if the secure element does not provide one.

Promise<Channel> openBasicChannel()

This methods opens a basic channel to communicate with a secure element application. Once this channel has been opened by an application, it is considered to be "locked" to other applications: any other call to this method will fail with `SENoChannelException` error until this basic channel is closed. Some secure elements might always deny opening a basic channel.

If the `aid` parameter is not null, the underlying implementation of this operation MUST send to the secure element a SELECT command, as defined in [[!ISO7816-4]], with following header values:

- CLA = ‘0x00’
- INS = ‘0xA4’
- P1 =’0x04’ (Select by DF name/application identifier)
- P2 =’0x00’ (First or only occurrence)

If `aid` is null, then no SELECT is sent, the channel is opened on the default selected secure element application.

This method will trigger the Access Control Enforcer to check the requesting application is authorized to open such channel.

Uint8Array aid

This parameter value MUST either be:

- The complete Application Identifier of the targeted application on the secure element;
- A partial Application Identifier matching a set of targeted applications on the secure element, in which case the channel will be opened on the first matching application;
- `null` if the channel should be opened on the default application.

Promise<Channel> openLogicalChannel()

This methods opens a logical channel to communicate with a secure element application. It is up to the secure element to choose which logical channel will be used. If no more logical channel is available, this method MUST fail with `SENoChannelException` error.

If the `aid` parameter is not null, the underlying implementation of this operation MUST send to the secure element a SELECT command, as defined in [[!ISO7816-4]], with following header values:

- CLA = ‘0x01’ to ‘0x03’, ‘0x40 to 0x4F’ (as chosen by the secure element)
- INS = ‘0xA4’
- P1 =’0x04’ (Select by DF name/application identifier)
- P2 =’0x00’ (First or only occurrence)

If `aid` is null, then no SELECT is sent, the channel is opened on the default selected secure element application.

This method will trigger the Access Control Enforcer to check the requesting application is authorized to open such channel.

Uint8Array aid

This parameter value MUST either be:

- The complete Application Identifier of the targeted application on the secure element;
- A partial Application Identifier matching a set of targeted applications on the secure element, in which case the channel will be opened on the first matching application;
- `null` if the channel should be opened on the default application.

Promise<void> close()

Closes the connection session to the Secure Element. This will close any channels opened by this application with this Secure Element. Invoking `close()` method on an already closed session is an idempotent operation.

L.15. CHANNEL INTERFACE

A Channel represents an [[!ISO7816-4]] channel opened to a Secure Element. It can be either a logical channel or the basic channel. It can be used to send commands to a Secure Element application.

readonly attribute Session session

This attribute MUST return the session object from which this Channel object was created.

readonly attribute ChannelType channelType

This attribute MUST return this channel type.

readonly attribute SEResponse? openResponse

This attribute MUST return the secure element's response to the channel opening operation. It MUST be `null` if the channel was opened on the default secure element application (no AID was provided in the open channel operation).

Promise<SEResponse> selectNext()

Updates the targeted application of this channel to be the next one matching the partial Application Identifier passed when this channel was open. Invoking this method MUST fail with an `SEInvalidStateException` error if this channel was not open with a partial AID, or with an `SENoApplicationException` error if there is no next application matching that partial AID. In that case the application associated to this channel is unchanged. If a next application has been found and associated to this channel, this operation succeeds and returns the secure element's response. This response value MUST be assigned to the `openResponse` attribute.

Promise<SEResponse> transmit()

This method transmits a command to the secure element. The user agent MUST ensure the synchronisation between all the concurrent calls to this method: a command MUST NOT be sent to a secure element while a response is still pending on any channel from this same secure element.

This method will trigger the Access Control Enforcer to check the requesting application is authorized to send such command on this channel.

The channel information in the class byte in the APDU command will be ignored. The system MAY modify the class byte of the command to ensure the APDU is transported on this channel. To ensure the invoking web application does not exit from the scope of this channel, the user agent MUST reject the following commands with `SEInvalidValueException` error value:

- MANAGE_CHANNEL (INS=0x70)
- SELECT by DF Name (INS=0xA4 and P1=04)

SECommand cmd
The command to send to the secure element application.

Promise<Uint8Array> transmit()

This method behaves exactly as the transmit method above, excepts that both its parameter and the response are passed as a raw binary data. Before transmitting the command to the secure element, the implementation of this method MUST set the logical channel in the class byte of the command so that it fits the channel allocated to this Channel object.

This method will trigger the Access Control Enforcer to check the requesting application is authorized to send such command on this channel.

Uint8Array cmd
The raw command to send to the secure element application. The channel information in the class byte of the command (first octet of the cmd data) will be ignored.

Promise<void> close()
This method closes this session object. If a transmit operation is still waiting for the secure element response, the user agent must terminate that asynchronous operation with a `closed` error status. Invoking `close()` method on an already closed channel is an idempotent operation.

L.16. CHANNELTYPE ENUM

The ChannelType enum identifies the type of the secure element channel.

basic
Basic channel, as defined in [[!ISO7816-4]] (channel number 0).

logical
Logical channel, as defined in [[!ISO7816-4]] (channel number > 0).

L.17. SECOMMAND INTERFACE

The SECommand interface represents an APDU command that can be sent to a secure element.

attribute octet cla
Class byte

attribute octet ins
Instruction byte

attribute octet p1
First octet of the parameter bytes

attribute octet p2
Second octet of the parameter bytes

attribute Uint8Array? data
Data field bytes, or `null` if command has no data

attribute unsigned short le
The length of the expected response data, or `-1` if application does not require a specific length

L.18. SERESPONSE INTERFACE

The SEResponse interface represents an APDU response received from a secure element.

readonly attribute Channel channel
This attribute MUST return the channel object that was used to transmit the command which triggered this response object.

readonly attribute octet sw1
First octet of response's status word

readonly attribute octet sw2
Second octet of response's status word

readonly attribute Uint8Array data
The response's data field bytes

boolean isStatus()
Utility method to test the status word of an APDU response. This method MUST return `true` if the parameters match the value of the response.
octet sw1
Value to compare to the first octet of response's status word, or `null` if this first octet may have any value.

octet sw2
Value to compare to the second octet of response's status word, or `null` if this second octet may have any value.

L.19. ERROR TYPES

In the interfaces defined above, some method return a Promise object. If an error occurs during the execution of any of these methods, the `reject()` method of promise's resolver will be invoked with an error value of one of the following DOMException subtypes:

SESecurityException
The requested operation does not match the access conditions of the application, as defined in Access Control section

SEIoException
Communication error

SEInvalidStateException
The target object was not in the proper state to execute the operation

SEInvalidValueException
The method was invoked with an incorrect parameter value

SENoChannelException
Tentative to open a channel failed because no channel is available

SENoApplicationException
The requested application was not found on the secure element

SEClosedException
The operation could not be fulfilled because the target object is closed

SEUnknownException
Internal error, no further details available

L.20. CODE EXAMPLE

The javascript code excerpt below shows how a web application can wait for a card to be present, and send it an APDU command:

```
// my application identifier
var myAppId = ...;
// my application command
var myAppCmd = new SECommand(...);

// get secure element manager
var seMgr = navigator.secureElementManager;

// register sepresent event handler
seMgr.onsepresent = function(reader) {

  // open session
  reader.openSession()
  .then(
    function (session) {
      // open a basic channel to my application
      return session.openBasicChannel(myAppID);
    }

  ).then(

    function (channel) {
      // open a basic channel to my application
      return channel.transmit(myAppCmd);
    }

  ).then(

    function (response) {
      if (!response.isStatus(0x90, 0x00) {
        // this might be an error
        // ...
      }
      response.channel.session.reader.close();
    }

  );
};
```

For simplicity and readability reasons, the code above omits the error handling that would have to be done in a real application.

L.21. CHANGES

The complete list of changes can be viewed on Github. You can also check the issues.

L.22. ACKNOWLEDGEMENTS

Thanks to contributors and reviewers...