RTC Bricks: Web Components for WebRTC Creativity

Kundan Singh
Intencity Cloud Technologies
San Francisco, CA, USA
kundan10@gmail.com

ABSTRACT

We present *RTC Bricks*, a collection of more than fifty web components to build multimedia, communication and collaboration applications using WebRTC (web real-time communication) and related technologies. These application use cases are demonstrated in over 150 sample web apps that cover a wide range of scenarios including audio/video call, multiparty conference, text chat, content share, device selection, click-to-call, one-to-many broadcast; shared tools such as whiteboard, notepad, or rich text co-editing; video conference layout such as flexible boxes, or 3D projections; video and image processing such as video effects, virtual or blur background, synthetic video, or water marking; or audio processing such as spatial audio, speech recognition, or text-to-speech. A separate native app included in the project allows web developers to create hybrid apps utilizing native non-web features such as raw network sockets, and frameless or transparent windows. These flexible, extensible and reusable web components are written in vanilla JavaScript with W3C recommendations, in about 40 kilo-source lines of code, and are not tied to any client side framework. The abstractions identified in our software are useful to other WebRTC-based web apps related to multimedia, communication and collaboration, and can be easily integrated with web developer platforms.

Keywords

WebRTC, audio/video conference, collaboration, endpoint driven, web component, JavaScript, audio/video effects, call signaling, telephony, video layout.

1. INTRODUCTION

Existing web multimedia communication or collaboration applications often suffer from one or more of these issues:

Locked to a JavaScript framework: Web developers often pick the best or most shiny framework at the time of initial development. Soon, the framework gets old, newer versions are invented breaking compatibility, and the app inherits a maintenance nightmare, or a steep upgrade cost [1][2][3]. Moreover, getting newer developers excited about working on the now old framework becomes challenging.

Tightly coupled to a service: Business incentives and fear of competition makes the client app locked to a single cloud hosted service [4][5][6][7]. Thus, every application ends up reimplementing the similar logic for common features such as video layout, text chat, meeting notes, or video effects.

The components are not truly reusable: The client app implementation is often component-based, but not reusable in general, because it gets locked to a single framework's component model, or to a single vendor's service [8].

Rigid user experience or one-size-fits-all: Product decisions often cater to popular workflows and use cases, while leaving out or disregarding niche features or controls [9][10][11]. The end user has little or no control over the product experience, e.g., to select how the text chat appears, whose webcam is shown, or how the videos are displayed.

We present *RTC Bricks* [12][13] to address these. Our project has a collection of more than fifty web components covering a wide range of multimedia, communication and collaboration use cases. These components are created in vanilla JavaScript [14], inline with W3C recommendations for web components and custom elements [15]. They do not use any client-side framework, and are independent of any single service. They are truly reusable as demonstrated in our sample apps. For example, our video conferencing user interface component can be configured to show either a text-focussed instant messenger or a video-first conference interface, among others. It can be replaced with a 3D layout display, or attached to a spatial audio component, if needed, to enable those features.

The components and their example use cases are detailed in a comprehensive live tutorial and developer guide [13], that prints to about 380 pages. It supports building hybrid apps, and can use non-browser features using a separate installed native app as part of this project. It also includes a couple of lightweight servers for signaling negotiation and real-time database, and allows interoperable connections to existing popular media servers and hosted services. Our project enables rapid prototyping of web collaboration applications, keeps the app logic separate from any user data, and promotes innovation in the endpoint.

The rest of the paper is organized as follows. Section 2 provides background and related work. The main categories of apps demonstrated in our project, and the software architecture of the web components used are described in Section 3. Section 4 has some implementation insights and metrics of the components, which are listed in Appendix A. Section 5 covers our conclusions and future directions.

2. BACKGROUND AND RELATED WORK

Web applications run in the browser, and are implemented using HTML, CSS and JavaScript. In the last decade, explosive growth in web applications gave rise to a plethora of JavaScript libraries and frameworks to address certain shortcomings in web standards [16]. Many of these have become obsolete, or are replaced by their standards-based counterparts in recent years [1][17]. Frameworks do offer benefits in specialized use cases, e.g., Vue is very easy to get started on UI-focussed single page apps, and HTMX is great for dashboards. However, unlike a library, a framework imposes huge technical debt after it becomes obsolete, because a framework is tightly intertwined with the app and vice-versa. Replacing it is akin to changing the skeleton while trying to preserve the functionality. Even upgrading the same framework often breaks backward compatibility, or requires a complete rewrite of the web app, e.g., upgrading from Angular version 1 to 2 [18].

Web components [15], on the other hand, are part of W3C standards [19], and are natively supported by web browsers. They are a set of web features such as custom element, shadow DOM and templates to create reusable custom HTML elements for modularity, encapsulation and interoperability across different web applications. In the past decade, while numerous frameworks kept appearing, these W3C component technologies matured, and are now fully supported on all popular web browsers. On one hand, popular frameworks are adopting some of these techniques, and on the other, emerging web applications are embracing web components instead of being chained to a single framework ecosystem.

WebRTC or web real-time communication [20] refers to an effort by W3C, IETF and popular browser vendors to allow web pages to exchange real-time media streams using a point-to-point media path. It includes several abstractions in JavaScript such as for media capture, device enumeration, peer-to-peer connection, media stream and data channel. Since its inception more than a decade ago, it has gained wide traction, and is supported on many popular browsers. However, the related WebRTC APIs are quite low level, and the existing web communication and collaboration apps based on WebRTC end up creating their own higher level abstractions, e.g., conference room, participant video box, or video layout. Instead of creating generic and reusable components, the past trend at such abstractions are often locked to a single vendor service [21][8][6][26][27][28][44], e.g., a click-to-call widget using telephony provider X, or a video conference one using service from vendor Y.

Our recent effort [22] broke this trend by creating a generic, flexible and reusable web component, video-io, for publish-subscribe of WebRTC media streams, without depending on any specific vendor service. It demonstrated a range of audio/video communication use cases including two-party, multi-party and one-to-many broadcast, using a named stream abstraction for media path. As we worked through many sample and demonstration web apps using this <video-io> component, we kept creating more abstractions and web components, e.g., for layout of multiple video boxes, text chat, speech transcription, or call signaling. The collection of these service independent generic and reusable web components became *RTC Bricks* [12][13], and they follow these four basic principles:

1. Separate any app logic from any user data.
2. Keep the app logic in the endpoint, if possible.
3. Use plain JavaScript; avoid any client side framework.
4. Create flexible and extensible custom elements.

Our work is the first attempt in the industry at providing a comprehensive collection of HTML5 web components for creating multimedia communication and collaboration apps, independent of any client-side framework, and more importantly, independent of any vendor locked services. The expansive list of supported use cases in our project are covered in the next section.

3. SOFTWARE ARCHITECTURE

The web components in our project can be classified into seven categories based on the application use cases they enable. These are shown in Fig.1, and discussed below. Some fall under multiple categories, e.g., media-chat is in both layout and collaboration tools.

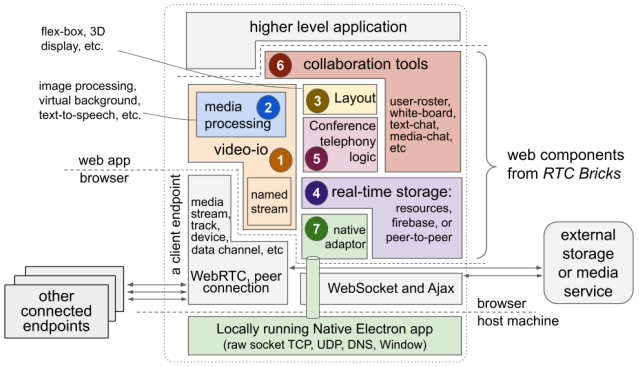


Fig. 1 Schematic diagram of our web components' categories, and their interactions with external entities.

3.1 Publish Subscribe of Media Stream

The video-io custom element (web component) combines various media and connection abstractions available in WebRTC, and exposes a single video box abstraction [22]. The component can be attached to a named stream, and configured in either a publish or subscribe mode as shown in Fig. 2. It demonstrates many use cases, e.g., live camera view, recording of multimedia messages, live video call or conferencing, in client-server as well as peer-to-peer topology. It encapsulates a standard HTML video or audio element, a WebRTC media stream (MediaStream), and zero or more connections (RTCPeerConnection) [20].

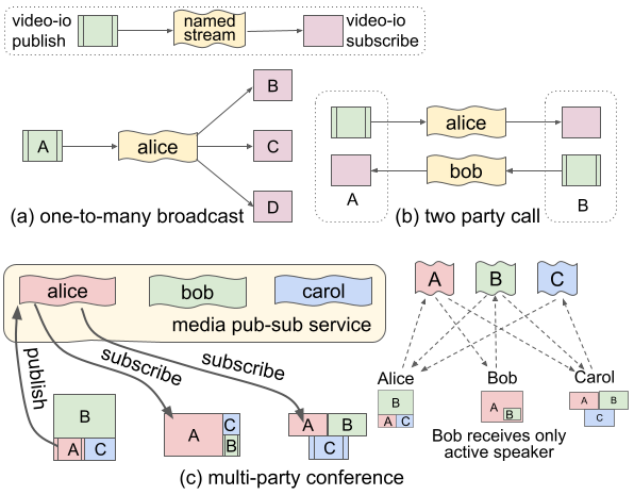


Fig. 2 Interaction between video-io and named-stream elements.

It hides the complexity of the underlying connection, device or media related functions, and exposes an extensive but easy-to-use API with more than 80 properties or attributes to control its behavior, or to receive indications of its internal state. This API allows a web app to easily send or receive a media stream, and create a range of app scenarios such as two-party call, multi-party audio/video conference, live event broadcast, or video presence in only a few lines of HTML and JavaScript code.

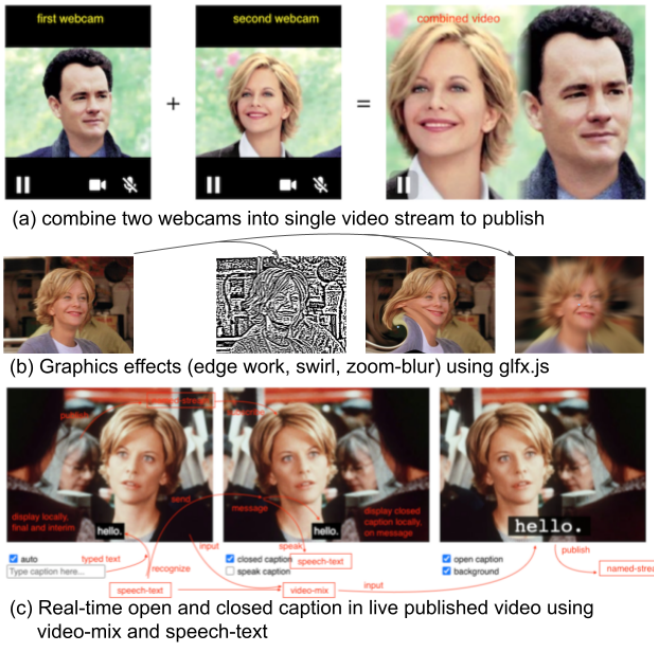
It uses a named-stream component abstraction, which is useful in many scenarios beyond just a call or conference. The vendor specific access control or signaling negotiation are present in a service specific derived component, e.g., rtclite-stream or firebase-stream. Besides the popular media servers such as Janus, FreeSwitch and Jitsi [24][25][26], and video conferencing services, e.g., Agora and LiveKit [27][28], we also support signaling and control using a real-time database, and custom connectivity to any existing or future media servers or video conferencing services.

The video-io component allows an app developer to write once, and be able to run the web app on different servers or services. Many client side features are included, e.g., select-zoom or magnifying glass on the video, customized controls, end-to-end media encryption using insertable streams, local client side recording, and delayed capture and playback for live captioning or moderated event broadcast.

These components readily work with any web app, installed progressive web app (PWA) [29], or browser extension. We also have a standalone web app, to use and experiment with existing or new named streams. It includes a flex-box component showing zero or more video-io instances, each attached to a named stream for publish or subscribe. In addition to the PWA for desktop, Android and iOS, we also include an installable Electron app [30] for desktop. It differs from the PWA in its launch URL, and requires a separate app logic to enable screen or window sharing.

3.2 Media Processing

The video-mix custom element receives media input from multiple sources (audio, video or video-io element), and uses an internal canvas element to programmatically combine them to generate output video frames. It can also mix or forward audio from the input. The output media can



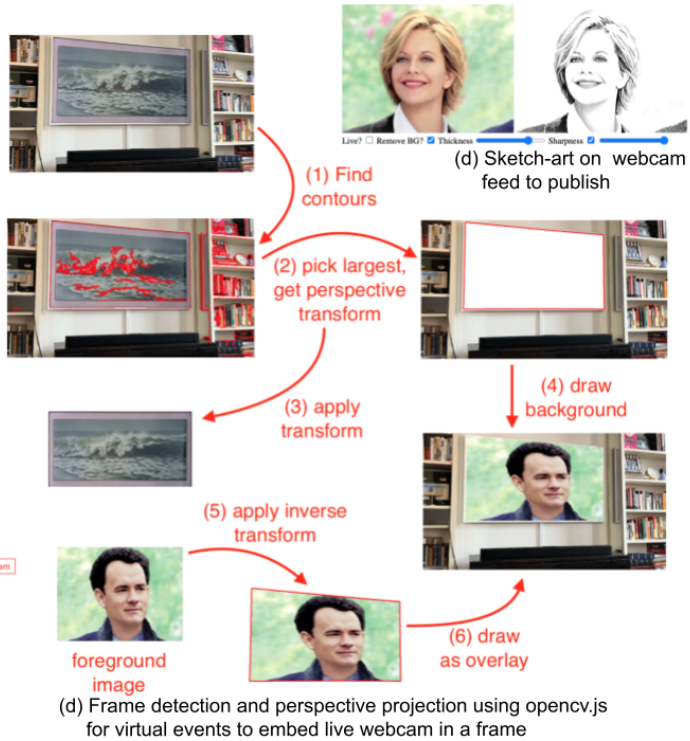


Fig. 3 Image processing on live webcam video to publish. (These illustrations use the RTC Camera and Screen browser extension [23] to use public celebrity images as webcam feed.)

then be used in lieu of video-io to publish or display in a conference. This enables a lot of endpoint driven creativity, as demonstrated and included in our sample apps listed below. Some of these are also illustrated in Fig. 3, using a virtual camera based on the RTC Camera and Screen browser extension [23].

Image and video manipulation: Displaying watermark or logo overlay on live webcam or shared content during a presentation. Sending media from an uploaded file to share live in a call. Altering image properties such as brightness, contrast, saturation or blur.

Combining multiple sources: Overlaying presenter webcam view on her screen share video. Combining multiple videos into a single video stream for recording or video mixing at the endpoint.

Temporal changes: Changing video speed by capturing webcam and displaying in slow motion, or fast forward. Capturing photo snapshot in slow motion for funny effects.

Image processing using external tools: Creating live sketch art from video using edge detection, color dodge, image sharpening. Using Tensorflow, BodyPix and/or Mediapipe [31][32] for background detection, blur, removal, or virtual background. Using face, eye and/or mouth tracking for auto-framing [8][33]. Using p5.js library for overlay effects, e.g., snowflakes, particles, rain, explosion, or fire [34]. Using glfx.js for graphics effects such as vibrance, denoise, sepia, swirl, pixelate, lens-blur, or dot-screen [35]. Using the opencv.js library [36] to do perspective projection and rectangular frame detection for virtual events with live feed projected on a virtual TV or frame. Using pixelmatch.js to detect changes in video frames, for security camera, which records the video only during a significant change in the captured images [37]. Using tesseract.js for OCR (optical character recognition) [38], for automatically taking notes from shared screen or content.

Audio signal processing using the Web Audio API: The audio-context custom element encapsulates various audio processing functions such as gain control, filters, and analyzers [39]. It can be attached to video-io or other elements, and used for audio activity detection, waveform generation, as well as spatial audio.

Speech recognition and generation: The speech-text custom element provides the speech recognition and text-to-speech functions [40]. It can be attached to video-io, media stream, or other elements, and used for automatic note taking. Together with video-mix, it is used for automatic closed or open caption overlaid on video in a call or conference. It uses the browser's built-in features for these, and supports many spoken languages in the Chrome browser.

These components hide the underlying complexity of media processing, and expose an extensive yet easy-to-use API of properties, attributes, events, and methods, inline with web component recommendations.

3.3 Video Layout

A flexible, adaptive and space optimized video layout in a conference user interface is important [41]. The existing CSS attribute, display, with flex or grid, is powerful, and can be used to achieve a good layout. However, they often need to be tweaked for a video conferencing user experience.

The flex-box component is a generic container to display multiple contained videos, images or other items. It allows customizing the display using a few attributes, while catering to a range of display scenarios suitable for video conferencing. An item may optionally be set as float, which is displayed larger than all the other items in the conference, e.g., for screen share, or active speaker's video. At a high level, this web component supports six layout variations as follows, and compared in Fig. 4.

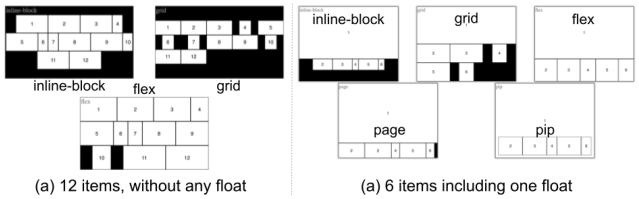


Fig. 4 Comparison of various display values in the flex-box layout

Inline-block: Items are in their natural sizes, scaled to fit the same height, while reducing empty space. A float item, if present, is attached to one of the four sides of the container.

Flex: Items are altered to change the aspect ratios within a limit, to fit the same height, and to minimize the empty space. An optional attribute controls whether object-fit of contain or cover is used. A float item, if present, is attached to one of the four sides.

Grid: Items are laid out in a grid, each with the same size of the bounding box, while reducing the outside empty space. A float item can occupy multiple rows and/or columns.

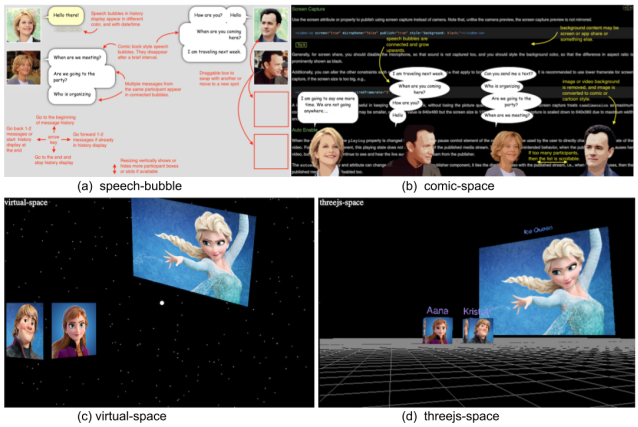
Picture-in-picture: One float item appears in the background in full size, whereas all the other items are overlaid in a line near one of the four sides. Scrolling of non-float items is enabled if their minimum relative size is set.

Page: A minimum size for each item is preserved, and the items can overflow to multiple pages, viewable by scrolling.

Line: This is like a scrollable page display, but with either one row or one column – the items appear in a horizontal or vertical line, for a film-strip or carousel-style layout.

The contained items may be in portrait or landscape orientation, with a desired range of aspect ratios for scaling. The flex display gives a fuller experience, while reducing the empty space not occupied by any item. The page display is suitable for a large number of items. Besides the display attribute, some other properties of the container or contained item control the other aspects, e.g., the animation on item add or remove, padding versus scaling within the item box, or aspect ratio flexibility or rigidity of the item.

The component also supports many user triggered controls, e.g., a double-click to change the float item, drag-move to re-position a non-float item or to attach a float item to a side, click-drag to resize the float versus non-float area, or scroll all the items depending on the display attribute.



(a) speech-bubble (b) comic-space

(c) virtual-space (d) threejs-space

Fig. 5 Video conference interface beyond a simple box layout

Although the box layout is the most common and popular video conference layout, we also support other kinds of layouts (Fig. 5). In particular, the speech-bubble container component implements a multi-party layout where each participant appears in a tiny box, and speech bubbles are used to show typed messages or spoken text after doing speech recognition. The comic-space container, on the other hand, displays the participant snapshot converted to comic book style characters, with speech bubbles popping above. These components can show shared content in the background, and allow scrolling back to see the message history, like navigating a comic book.

We also include two containers that display in 3D layout. The threejs-space element uses the popular three.js library [42] for rendering the items on a canvas. The virtual-space element uses CSS perspective transform to display the items in 3D more efficiently. For audio, the spatial-audio component can be attached to a flex-box container, and audio-space can be attached to virtual-space, to quickly enable spatial audio based on the contained items positions in those containers.

All these layout containers have similar semantics for adding or removing any contained items. The flexibility and reusability of our video layout approach is demonstrated by dynamically changing the box layout display, or changing it to a comic book style layout or 3D rendering in real-time, driven entirely by the end user in the endpoint, without any changes in the conference or service logic in the web app. Newer layout behavior can be easily created as well.

3.4 Real-Time Shared Data Storage

Although the video-io component supports real-time data exchange using the built-in data channel, it is not enough for many application use cases. In particular, real-time data storage is needed for the participant list, user presence status, text message history, and the state of various collaboration tools.

The ability to access and modify shared data is crucial in a distributed application. Furthermore, the clients can get notified when a piece of shared data is modified, and can update the client's display state. For example, participants' list or user's presence information can be stored as shared data, and the user's client can modify and get notified on any data change, to implement the entire app logic in the endpoint. This resource-oriented software architecture has been well researched in our previous work [43][44], as well as the popular Firebase real-time database [45].

The shared-storage component provides an abstraction for such real-time data storage. The data or resources are stored hierarchically, identified by a path similar to file path. A path can refer to a specific object, or a list of objects. The create, read, update and delete operations are supported on the object path, and the add, get all and remove all on the list path. Additionally, these paths can be subscribed for a change event, such that any change is delivered to both the object path as well as the parent list path. Any notification message may also be sent on any path, which are delivered to any clients subscribed on that path for the notify event. This small set of methods and events on the resources are enough to implement a wide range of communication and collaboration applications in our project.

Our abstraction also supports creating a transient resource, that is automatically deleted when the creator client is disconnected. Alternatively, a transient resource may be bound to the parent's list path, in which case it is removed when all the subscribers to that list are disconnected. These concepts are used in implementing many features such as user presence, conference membership, or ephemeral text chat history that is automatically cleaned up when all the participants leave.

The abstract shared-storage component must be configured to use an external storage in a real application. Thus, a local demonstration app is quickly turned into a real app by just replacing the storage component. For example, local demo apps use the browser's localStorage, but an actual app uses restserver-storage, firebase-storage, peer-storage, or other, depending on which backend storage service is desired. The restserver-storage connects to a resource server based on our earlier work, and is available in Python, PHP and NodeJS, with a backend database of in-memory, SQLite, MySQL or PostgreSQL [46]. The firebase-storage web component connects to Google's Firestore database [45], and implements the missing transient resources and message notifications using its existing features. The peer-storage component uses a peer-to-peer network for data storage and message notifications [47].

We implement endpoint driven redundancy and scalability

of data storage. The redundant-storage component enables storage redundancy for reliability and failover. It includes multiple other storage elements, writes to all, and reads from any. The partition-storage component uses sharding of data among the included storage elements. The sharding is based on a simple hash of the resource path of the list, and requires that list and object resources alternate in the path, so that the list and its objects are handled by the same partition. These components demonstrate the flexibility of the software architecture and the component model.

The storage-stream component uses the shared-storage component to implement a named stream abstraction for publish-subscribe of media streams. This allows a single service for both the shared data and named streams, e.g., when using the restserver-storage component. The shared-storage component is used for telephony call signaling, conference join flow and membership, as well as for storing and exchanging data in various other collaboration tools.

3.5 Telephony and Conference Control Logic

Although the video-io and named-stream components are enough to enable various call and conference use cases, a lot of app logic is involved in call signaling or conference join workflow [41][44]. For example, in a two-party call (Fig. 2b), call signaling can be used for three things: (a) to register a user name, so that others can discover and reach this user, (b) to send an intent to call, so that the receiver can answer or decline, and (c) to exchange the stream names so that both the endpoints can launch their video-io components attached to the right named streams. The detailed steps are different if an early media is desired, or if multiple lines (or user devices) are allowed for a user name. For a conference, with join-leave flow, the shared storage includes the unique member identifiers and membership data. The call signaling can be combined with the join-leave flow to create an invite-answer flow where an user is invited to join a conference, but leaves on her own. In such cases, multiple lines and early media become relevant again.

The phone-state component implements a customizable state machine for phone registration and call invite, and the conference-state component implements that for conference join and leave, and maintains a list of active members in the conference. These have easy-to-use attributes to control various intricate behaviors. Apps that use a conference or phone state machine also need to add or remove video-io instances when a call is connected, a conference is joined, or another member joins or leaves a conference. This is done in the videos-control component. It acts as a controller to connect a phone or conference state machine with video-io elements, and can optionally attach to a layout container such as flex-box for video layout.

A number of telephony related applications use a few well defined scenarios, such as click-to-call, click-to-join, or sequential invites, parallel call distribution, or call queues. The general purpose click-to-call component supports many such use cases including click to call or answer, click to join or leave, queue incoming calls, or distribute outbound calls. It includes zero or more instances of phone-state or conference-state. At least one state component is needed for proper functioning.

The number and type of the state components, together with some other attributes, determine the behavior of the click-to-call component in various scenarios, e.g., a single phone device to send or receive a call, a single conference attendee to join or leave a conference, a conference attendee that can also send or receive an invite to join a conference but leaves on its own, a call queue to postpone incoming invite if already in a call, and to answer any call from the queue, or an outgoing call distributor to multiple targets in sequence or parallel, with or without timeouts. Depending on the component behavior, some constraints are imposed or features enabled, e.g., a missed call is not applicable in a call queue, and a phone call requires that receiver to support video if the caller initiates a video call, whereas an invited conference join allows mismatch in media capabilities.

3.6 Collaboration Tools

Many collaboration tools are included in the project, and are described here. Such tools are implemented as a web component for its interface and app logic, and optionally, another for its data model. The data model component is attached to a shared-storage component for real-time data storage, notification, and state sharing. Separating the user interface from the data model allows replacing the data with some other storage in the future, while preservice the app logic in the tool [44]. Some components can work with or without a data model, e.g., shared-editor can act as a text chat input tool without a data model, but turns into a shared and synchronized rich text editor with a data model.

Text Messaging

To enable simple text chat among collaborating users, there are two components: text-feed and text-chat, and one data model, text-feed-data. The text-feed component displays the real-time text feed of a shared list of messages. The text-chat component then uses this, along with a text input area, typing indication and drag-and-drop file share feature, to provide a full text chat and messaging support among the collaborating users. This component also supports popular text chat features such as customizable smileys or emoticons, alert sound on new messages, or clickable link detection. Data attributes such as timestamp, disposition and sender information are used to display the message differently as applicable. Additionally, rich text using HTML format is supported, if enabled. Optionally, editable messages are allowed, where the user can edit or remove a previously sent message. The exact behaviors are customizable, e.g., to remove or strike-out a deletion.

The text-feed-bubbles component is similar to text-feed,

except that it shows the chat messages as speech bubbles, instead of a simple continuous text area, similar to other popular text chat applications (Fig. 6a). Additionally, it includes child elements of the text-item-bubbles component, one per message. This allows better styling or replacing the individual message display, if needed. When this component is included in a text-chat instance or attached to a storage path, it adds the child text-item-bubbles elements automatically based on the messages received or sent on the storage path. These components are highly customizable, e.g., bubble style, position, color, font and so on. The text input area can be replaced with a rich text editor such as our shared-editor component, for a formatted HTML text input.

Our project also includes some web components to display the text chat user interface similar to the popular Slack enterprise messaging client, including its light versus dark theme, display mode of feed, thread, or compact, uploaded content share, sender icon versus name, and grouped replies. These attach to the same data model as other text chat and feed components, and demonstrate the flexibility of the component model.

Collaborative applications often need the ability to display a list of users along with their attributes or capabilities. For example, a contact list in a messenger application or an attendee list in a conference application show the list of users along with some audio or video indication. Our user-roster component achieves this, and attaches to its data model component, user-roster-data.

Multimedia Chat

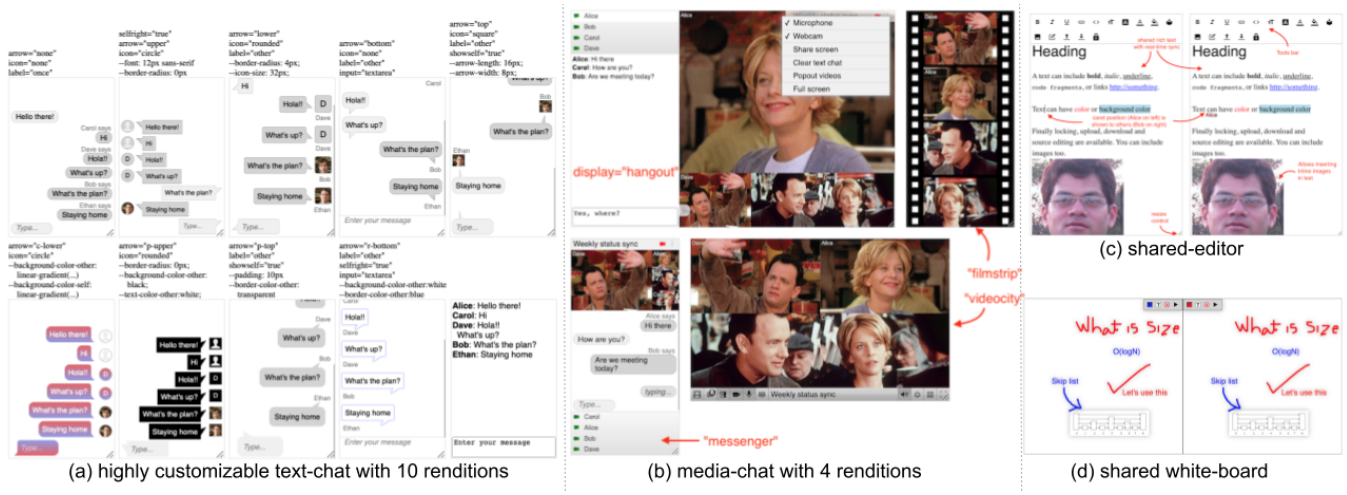
The media-chat web component enables a full fledged multimedia chat, and includes other components to perform text chat, display attendee list, and optionally, participant videos. Separate instances of the component can connect to the same path on the shared-storage to participate in a multimedia chat. The component uses a media-chat-data instance (data model), which in turn includes text-feed-data and user-roster-data as the data models for the included text-chat and user-roster, respectively. The paths of the included data models are derived from the top-level path, e.g., if the media chat's path is "sessions/conf123" then the path for the user roster becomes "sessions/conf123/users", text chat becomes "sessions/conf123/messages", and for the users' media becomes "sessions/conf123/calls" [41].

The user interface of media-chat is highly customizable, to cater to different use cases. The toolbar-buttons component can be used to display a toolbar, and the overlay-menu to display a programmable menu. Except for the toolbar or overlay menu, the order of the included components is important, and is preserved in the display. The component properties control how an active speaker is highlighted, or whether sound level is shown. It also supports private messages, custom commands, and synchronizing layout of contained display items among participants

The display property controls the overall layout. It can be set to messenger, hangout, filmstrip or videocity (Fig. 6b). The messenger display is similar to chat-first apps, e.g., the once popular Yahoo messenger, with focus on text chat, and shows an inline video box when needed. The hangout display is similar to video-first apps, e.g., Google hangout, with focus on videos, and the ability to show chat and user roster on the side. The filmstrip display is intended for narrow aspect ratio, and shows the video boxes in a film strip, which can be flipped back to the messenger style text chat and roster display. The videocity display is intended for embedding in web pages, and appears as a video player style interface, where all the included elements appear as boxes in the container [48]. More custom behaviors can be added, e.g., separate window for each collaboration tool, or combining video and text chat in a comic-space container.

Additional Tools

The white-board component implements a simple white board which can be attached to a white-board-data data



(a) highly customizable text-chat with 10 renditions (b) media-chat with 4 renditions (c) shared-editor (d) shared white-board

Fig. 6 Demonstrations of some collaboration tools related web components

model to enable shared white board (Fig. 6d). Besides drawings and text inserts, it also allows drag-and-drop of image files, and downloading content as an image file.

The locked-notepad component implements a text editor that can be locked for editing to avoid conflict. It can be attached to a locked-notepad-data instance to enable shared notepad. Only one user may have the editing lock at any time. The component also supports copy, cut and paste.

The shared-editor component, on the other hand, includes a rich text editor (Fig. 6c). Although this one does not require locking, it is still supported to avoid conflict. The real-time synchronization of the shared text currently has some inconsistencies during simultaneous editing, which can be resolved in the future using operational transforms. Like the previous component, this also attaches to a data model, shared-editor-data, and a shared storage to enable shared editing. In shared editing mode, it can also show other participants' cursor and caret.

Shared Games

The collaboration tools described previously such as for text chat, notepad or whiteboard are examples of endpoint driven applications [44]. The shared-storage data models used by these tools enable collaboration. Such collaboration is not limited to these use cases, but can be expanded to others that need shared state among participants such as for shared games. We include chess-board and ludo-board to demonstrate two games, Chess and Ludo, respectively.

Shared Apps

The media-chat component allows sharing text, audio, video, screen or an app window with other participants. It also has methods such as startshare, stopshare, addshare and removeshare, which are used to share any custom app such as for participants' survey to gather real-time responses from the attendees in a conference, or slide show of a photo album. Internally, a message channel is used to share any app data among the participants.

The slide-show component can display one or more image, video or PDF files with navigation controls, and does not need a shared-storage. It can be used with independent navigation at each participant, or with synchronized slide show or media playback. In this synchronized co-viewing mode, the component at the viewer end is controlled and navigated by the sharer, e.g., for page navigation, or video seek. If a PDF file is shared, then it uses the pdf-viewer component internally to display the file [49].

When a file is shared via the text chat or drag-and-drop in the text input area, the chat history data contains the full content of the file. When such a file share link is clicked in the text chat component, it dispatches the openurl event, which bubbles up to the application. The application can listen to this, and decide to use the slide-show component to show the file inline, instead of the default download.

To enable sharing of multiple webcams or screens from the same user, the app can wrap the corresponding video-io component in a shared app, and send supporting data such as its stream name to others, so that all the receivers can subscribe to its published stream. Other components such as white-board, shared-editor, or chess-board, can be shared similarly among the participants in a conference. The flexibility and loose coupling among the components enable such mash ups. Moreover, adding a new collaboration tool or extending an existing one using shared-storage is simple.

3.7 Native Features Adapter

For an installed app, outside the browser, a PWA [29] is better than an Electron app [30] due to its smaller size, and build simplicity. However, a native Electron can provide many additional features that are not available in a web app or PWA [50]. We include such a NativeElectron app, and provide a web component, native-electron, as an adapter to expose many new native features to the web app.

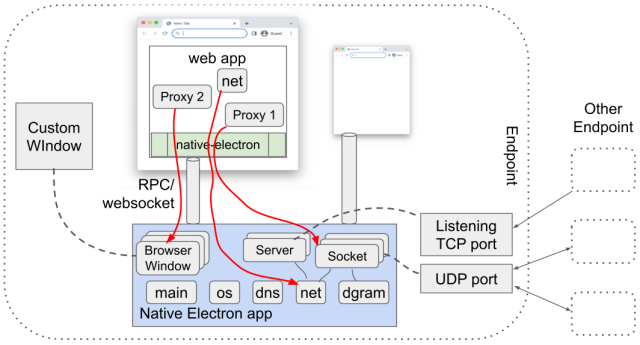


Fig. 7 The NativeElectron app and the native-electron adapter.

The architecture is shown in Fig. 7, where the native app acts as a plugin to enable and expose several Electron APIs, e.g., for opening a native window, getting system information, DNS resolver, and raw TCP and UDP sockets. The basic idea is inspired by the now obsolete flash-network project [51][52], that used a native AIR (Adobe Integrated Runtime) app to expose certain network APIs to web apps. In our project, the web app uses the native-electron custom element to connect over WebSocket to the native app. This allows the web app to use, say, raw sockets, for implementing various networking applications such as peer-to-peer or application level multicast, or for implementing custom window behavior, such as to open a transparent or frameless browser window for a more immersive video conferencing display, or for drawing on the screen or showing the mouse pointer of the other participants during a screen share enabled video collaboration. Note that such features are not possible in the web app or PWA alone using the current web technologies.

The actual set of features are controlled by the native app, and the adapter just exposes those APIs, proxies the method request, result, and any events, and performs any error handling, e.g., unsupported or unauthorized calls. Currently,

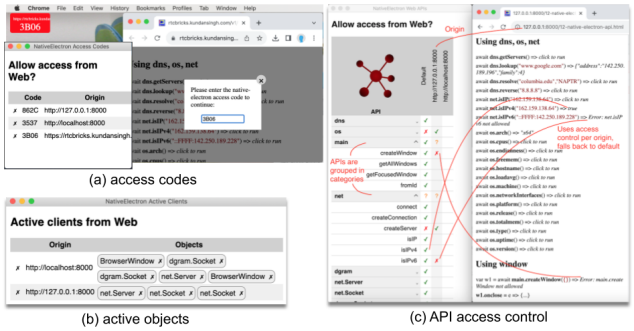
our native app supports five modules from Electron [30] and NodeJS: dns, os, net, dgram, and main (for native window creation and query). Additionally, it supports four types of objects, net.Server, net.Socket, dgram.Socket and BrowserWindow. These modules and objects have their own set of methods and events which are exposed from the native app to the web component, using the proxied objects or modules. Similar to any RPC or RMI (remote method invocation) protocol, our implementation takes care of using object identifiers, serialization and deserialization.

All the methods exposed on the above objects and modules by the adapter are asynchronous, even if the underlying native method is synchronous. Since property access in JavaScript is synchronous, the proxied objects or modules do not provide any direct property access, unless the property is accessed via a method call. All the proxied objects in the adapter receive all the relevant events from the native app, and depending on which event has a handler installed by the web app, only those are triggered by the web component.

The native APIs exposed by the adapter in conjunction with the locally running app are very powerful, and if misused, can cause lasting damage to the local system. Thus, such APIs should be allowed only from trusted web pages or apps. Note that some method results are redacted for privacy, e.g., the MAC address field is redacted when listing the network interfaces on the os module. Some methods have restrictions, e.g., a socket may only listen on a port, not a Unix path, and the port must not be lower than 1024 to avoid interfering with any local services. Additionally, the native app implements a simple authentication to ensure that a web app cannot connect to the native app without any user interaction or input for the first time. This mechanism keeps the end user in control of the authentication and authorization phase.

We support three layers of access control as follows. First, when a web page from an origin connects for the first time to the native app, a random passcode is generated, which is displayed to the end user, and must be entered in the web app to allow a connection from any web page on that origin. Second, for each origin, the end user can select which API methods are enabled or disabled, or configure the access control for the default origin. Third, for each web app, the end user can see all the active objects such as open sockets or windows, and can selectively force close them. These access control mechanisms are implemented in the native app, using native user interface for direct end user control of these features as shown in Fig. 8.

Our project includes several sample apps to test the native features such as DNS lookups, UDP and TCP sockets, and native frameless or transparent windows. One such app shows a video conference layout where the video of a participant appears in an always-on-top and draggable circle directly on the desktop screen, instead of within a browser or other window. Another app demonstrates drawing on the screen with mouse input, via a transparent full size window, and using SVG (scalable vector graphics) for the drawings.



(a) access codes (b) active objects (c) API access control

Fig. 8 Fine grained access control of native APIs by the end user.

The various web components described earlier are the building blocks of a wide range of applications. However, a specific application may need only a few of these blocks, e.g., native-electron is useful only in an installed app, and virtual-space is only needed when a 3D layout is desired.

4. IMPLEMENTATION

The implementation of various web components and their sample applications has been an incremental process. Many of the components such as video-io, shared-storage, rtclite-stream, and video-mix, have matured with several demonstration apps. Some are still primitive and need to be improved, e.g., redundant-storage and verto-stream. Our list of web components is still evolving, e.g., to address newer use cases, or to integrate newer services.

The current snapshot of these web components is shown in Fig. 9, and summarized in Appendix A. It also shows how the various components are related to each other, e.g., the peer-storage and restserver-storage components are derived from the base, shared-storage; many components in the service layer implement the named-stream interface; phone-state and text-feed-data use shared-storage; media-chat contains flex-box which in turn contains video-io; and text-chat uses text-chat-data as the data model. Since many of the collaboration tools are based on shared-storage, the base classes, ProxyDataElement and ProxyStorageElement, with common code, are used to create derived components related to the app logic and data model, respectively.

There are about 70 web components implemented in about 40 JavaScript files, with combined about 40 kilo-source lines of code (kSLOC), using the W3C custom element and web component guidelines, without using any JavaScript framework. The distribution of these component files is shown in Fig. 10a. Most of the files are very small. Only a handful are more than two thousand lines each. About 150 sample web app files are each relatively very small – most of them less than 200 lines of code (see Fig. 10b).

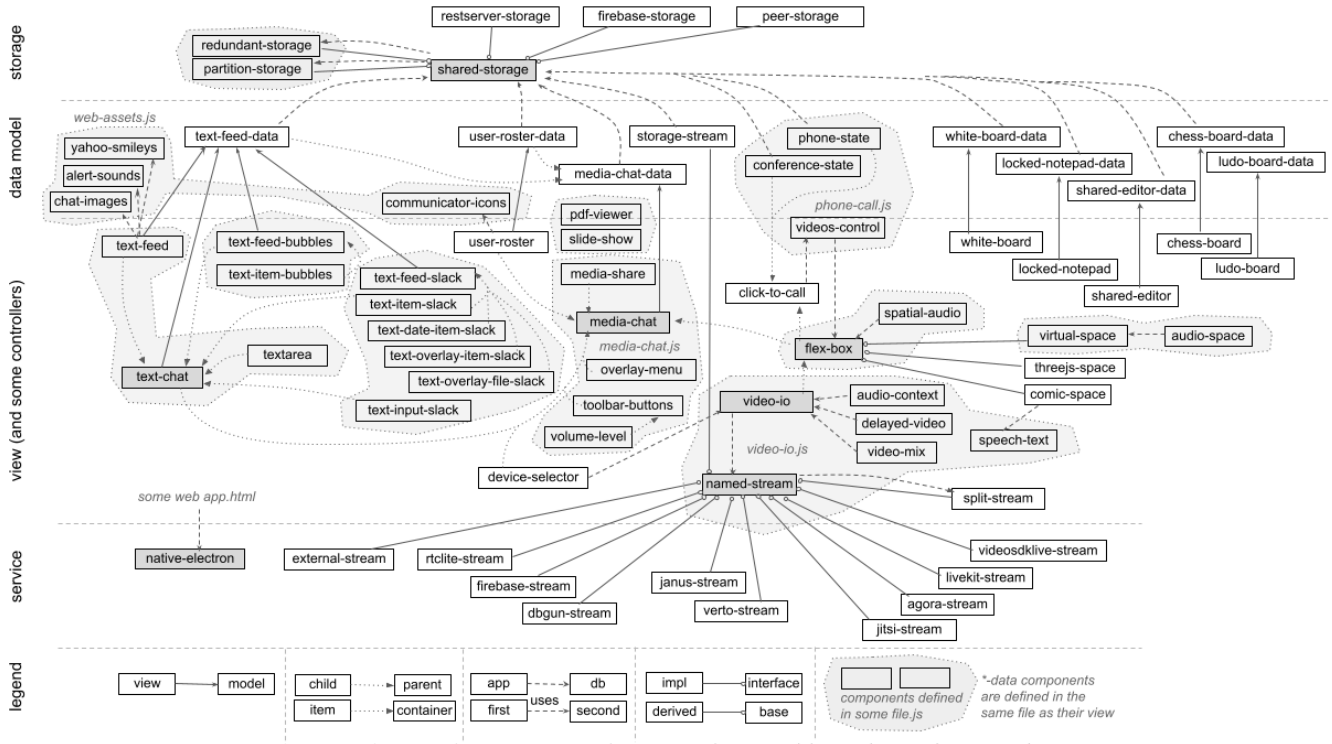
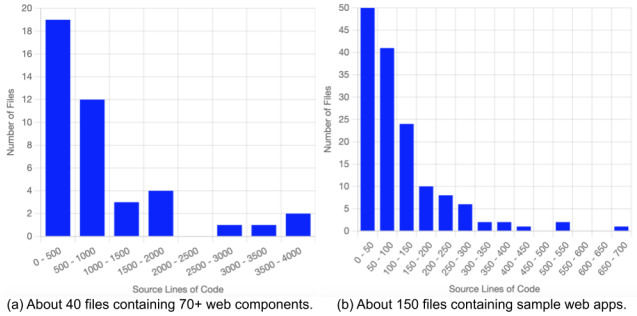


Fig. 9 Various web components in our project, and how they relate to others.



(a) About 40 files containing 70+ web components. (b) About 150 files containing sample web apps.

Fig. 10 Implementation complexity in terms of source lines of code of the various web components and the sample apps.

5. CONCLUSIONS AND FUTURE WORK

Our software [13] is still in its early stage, but convincingly demonstrates the flexibility, reusability and simplicity of using web components for a wide range of web multimedia, communication and collaboration use cases. The software architecture covers audio/video streaming, multiparty multimedia conferencing and collaboration, audio/video and image processing, innovative user interface for layout, and hybrid web apps with native capabilities. Many of the included web components are extensible, combinable, and replaceable, e.g., to easily switch to a different media server or real-time data storage, or a different text chat display.

Our component architecture, based on a few simple guiding principles, covers many common and existing video call and conferencing related product experiences. It addresses some nuanced use cases as well, e.g., with its collection of audio/video effects, and 3D projections. Innovation in the endpoint is further promoted by loose coupling among the components, and separation of user data from the app logic.

In the future, we plan to expand the list of web components, support cross browser compatibility, go beyond Electron for the native app, and optimize certain media processing logic. We are also creating a graphical drag-drop app builder that can allow compositing web components and other high level app logic to create customized web apps, or other complex web components. Our work promotes reuse of many media and collaboration features across web apps, reduces vendor lock-in, and provides an extensive collection of end-point driven web components for a large number of multimedia collaboration use cases.

6. REFERENCES

[1] Ambriz, J.C.A. “Front-end frameworks: solutions or bloated problems?” Blog Article. Jan 2026. https://www.toptal.com/developers/javascript/are-big-front-end-frameworks-bad

[2] Gregorio, J. “No more JS frameworks”, Talk at Open Source Software Convention (OSCON). 2015. https://bitworking.org/news/2014/05/zero_framework_manifesto/

[3] Pike, A. “A JS framework on every table.” Blog Article. 2015. https://allenpike.com/2015/javascript-framework-fatigue/

[4] Jeyaseelan, S. “Vendor lock-in issues in cloud computing and how to neutralize them.” 2025. ProQuest Thesis. ID: 3178461730, ISBN:979-8-31-012636-7

[5] Berners-Lee, T. (2010). “Long Live the Web: A Call for Continued Open Standards and Neutrality.” Scientific American, 80-85. Retrieved Dec 2025: http://www.scientificamerican.com/article.cfm?id=long-live-the-web

[6] Naik, V. “Open web vs. walled gardens: understanding the

Internet's ecosystems." Blog. Jan 2026. https://wpedition.com/open-web-vs-walled-gardens/
[7] Paterson, N. "Walled gardens: the new shape of the public internet." Proceedings of the 2012 iConference. 2012. DOI: 10.1145/2132176.2132189
[8] Singh, K. "Unlocking WebRTC for end user driven innovation." Technical report. arXiv. Dec 2025. DOI: 10.48550/arXiv.2512.23688
[9] Singh, K. "How not to design a video conferencing product." Blog. May 2024. https://blog.kundansingh.com/2019/06/how-not-to-design-web-conferencing.html
[10] Hove, D.P. and Watson, B. "The shortcomings of video conferencing technology, methods for revealing them, and emerging XR solutions." Presence: virtual and augmented reality. 2022. DOI:10.1162/pres_a_00398
[11] Toxboe, A. "Why one-size-fits-all behavior design fails." Blog. Feb 2025. https://learningloop.io/blog/why-ons-size-fits-all-fails
[12] Singh, K, "RTC Bricks." Apr 2026. Technical Report. DOI:10.13140/RG.2.2.18147.92969
[13] Singh, K. "RTC bricks: unleashing WebRTC creativity using web components." Project. Accessed Apr 2026. https://github.com/theintencity/rtcbricks
[14] Learn JavaScript. Project page. Accessed Apr 2026. https://github.com/snipcart/learn-vanilla-js
[15] Web components. Wikipedia. Accessed Apr 2026. https://en.wikipedia.org/wiki/Web_Components
[16] Comparison of JavaScript-based web frameworks. Wikipedia. Accessed Apr 2026. https://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_web_frameworks
[17] Tilgner, A. "Should you really be using a JS framework?" Blog Article, Medium. Oct 2021.
[18] Koch, P.P. "The problem with Angular." Blog. 2015. https://www.quirksmode.org/blog/archives/2015/01/the_problem_wit.html
[19] WebComponents. W3C Wiki. Accessed Apr 2026. https://www.w3.org/wiki/WebComponents
[20] WebRTC: Real-Time Communication Between Browsers, W3C Recommendation. 2025, http://www.w3.org/TR/webrtc/
[21] Levent-Levi, T. "What is vendor lock-in in WebRTC?" Dec 2025. https://bloggeek.me/vendor-lock-in-webrtc
[22] Singh, K. "A generic web component for WebRTC pub-sub." arXiv. Feb 2026. DOI:10.48550/arXiv.2602.22011
[23] Singh, K. "RTC Helper." Project. Accessed Apr 2026. https://github.com/theintencity/rtchelper
[24] Janus: general purpose WebRTC server. Project. Accessed Apr 2026: https://janus.conf.meetecho.com/docs/
[25] FreeSWITCH: open source telephony for voice, video and messaging. Apr 2026: https://signalwire.com/freeswitch
[26] Jitsi: open source video conferencing platform. Project. Accessed Apr 2026: https://jitsi.org/
[27] Agora Video SDK. Accessed Apr 2026: https://agora.io
[28] LiveKit Cloud SDK. Accessed Apr 2026: https://livekit.io
[29] Progressive web app (PWA). Wikipedia. Accessed Apr 2026. https://en.wikipedia.org/wiki/Progressive_web_app
[30] Electron JS: build cross-platform desktop apps. Project. Accessed Apr 2026. https://www.electronjs.org/
[31] Oved, D., Zhu, T. "BodyPix: real-time person segmentation in the browser with tensorflow.js" Blog. Nov 2019. https://blog.tensorflow.org/2019/11/updated-bodypix-2.html
[32] Grishchenko, I., et. al. "Body segmentation with MediaPipe and tensorflow.js" Blog. Jan 2022. https://blog.tensorflow.org/2022/01/body-segmentation.html
[33] Tracking.js: a modern approach for computer vision on the web. Project. Accessed Apr 2026. https://trackingjs.com
[34] P5.js: open project. Accessed Apr 2026. https://p5js.org
[35] Glfx: an image effects library using WebGL. Project. Accessed Apr 2026. https://github.com/evanw/glfx.js
[36] OpenCV JavaScript version for node.js and browser. Project. Accessed Apr 2026. https://techstark.github.io/opencv-js/
[37] Pixelmatch: pixel-level image comparison. Project. Accessed Apr 2026. https://github.com/mapbox/pixelmatch
[38] Tesseract.js: pure JavaScript OCR. Project. Accessed Apr 2026. https://github.com/naptha/tesseract.js
[39] Web Audio API 1.1. W3C working draft. Nov 2025. https://www.w3.org/TR/webaudio-1.1/
[40] Web Speech API. W3C. 2025. https://webaudio.github.io/web-speech-api
[41] Singh, K. "Vclick: endpoint driven enterprise WebRTC." IEEE International Symposium on Multimedia (ISM), Miami, FL, USA. Dec 2015.
[42] Three.js: JavaScript 3D library. Project. Accessed Apr 2026. https://threejs.org/
[43] Davids, C., et al. "SIP APIs for voice and video communications on the web." International conference on principles, systems and applications of IP telecommunications (IPTcomm), Wheaton, IL. Aug 2011.
[44] Singh, K. and Krishnaswamy, V. "Building communicating web applications leveraging endpoints and cloud resource Service." IEEE International Conference on Cloud Computing (IEEE Cloud), Santa Clara, CA, USA. Jun-Jul 2013.
[45] Cloud Firestore: a real-time database. Accessed Feb 2026: https://firebase.google.com/docs/firestore
[46] Vvowproject: voice and video on web. Project. 2011-2012. https://github.com/theintencity/vvowproject
[47] Singh, K. "Lightweight call signaling and peer-to-peer control of WebRTC video conferencing." Technical report. arXiv. Feb 2026. DOI:10.48550/arXiv.2602.08975
[48] Singh, K. "Videocity: for video telephony and conferencing". Project. 2009. https://github.com/theintencity/videocity
[49] PDF reader in JavaScript. Project. Accessed Apr 2026. https://github.com/mozilla/pdf.js
[50] Electron-socket: networking socket for Electron apps. Project. Accessed Apr 2026 https://github.com/foxglove/electron-socket
[51] Singh, K. "Flash-network developer guide". Project page. 2011. https://theintencity.kundansingh.com/flash-network/
[52] Singh, K. and Davids, C. "Flash-based Audio and Video Communication in the Cloud." Technical report, arXiv. Jun 2011. DOI:10.48550/arXiv.1107.0011

APPENDIX A: WEB COMPONENTS LIST

Table I Current list of web components or RTC Bricks.

Name	File name, and description
video-io	video-io.js
A video box for local or remote video, can optionally be in publish or subscribe mode attached to a named-stream.	
video-mix	video-io.js
Manipulate or create video using canvas and image processing.	
delayed-video	video-io.js
Playback of audio/video using internal recording and delay.	
audio-context	video-io.js
Apply Web Audio API on attached video-io or media stream.	
speech-text	video-io.js
Speech recognition and synthesis using built-in JavaScript APIs.	
device-selector	device-selector.js
Allow selecting devices, and optionally attached to video-io.	

named-stream	video-io.js
Basic abstraction of named stream, attached to video-io.	
rtclite-stream	rtclite-stream.js
Named stream using the RTCLite project.	
janus-stream	janus-stream.js
Named stream using the popular Janus media server (SFU).	
verto-stream	verto-stream.js
Named stream using the Freeswitch media server (MCU).	
firebase-stream	firebase-stream.js
Named stream using the Cloud Firestore service.	
dbgun-stream	dbgun-stream.js
Named stream using Graph Universe Node distributed database.	
jitsi-stream	jitsi-stream.js
Named stream using Jitsi-as-a-Service for conferencing.	
agora-stream	agora-stream.js
Named stream using Agora video conference service.	
livekit-stream	livekit-stream.js
Named stream using LiveKit video conference service.	
videosdklive-stream	videosdklive-stream.js
Named stream using VideoSDK.live conference service.	
external-stream	external-stream.js
Named stream using any third-party web app.	
split-stream	split-stream.js
Named stream using shared-storage.	
shared-storage	shared-storage.js
Abstraction for real-time data base of hierarchical resources.	
restserver-storage	restserver-storage.js
Shared storage using the restserver project.	
peer-storage	peer-storage.js
Shared storage using a peer-to-peer network.	
firebase-storage	firebase-storage.js
Shared storage using the Cloud Firestore service.	
redundant-storage	redundant-storage.js
Allows redundancy among shared storage for reliability.	
partition-storage	partition-storage.js
Allows partitioned data among shared storage for scalability.	
flex-box	flex-box.js
A flexible layout container which is highly customizable.	
threejs-space	threejs-space.js
3D layout container using the popular three.js project.	
virtual-space	virtual-space.js
3D layout container using standard CSS perspective transforms.	
speech-bubble	speech-bubble.js
Conversation tracking and layout container with speech bubbles.	
comic-space	comic-space.js
Like speech-bubble, but comic book style participant images.	
spatial-audio	flex-box.js
Spatial audio using 2D position, can attach to flex-box items.	
audio-space	virtual-space.js
Spatial audio using 3D position, can attach to virtual-space.	
text-feed	text-chat.js
Display of messages in real-time and/or from history.	
text-feed-data	text-chat.js
Data model for text-feed, text-chat, etc., using shared-storage	
alerts-sounds	web-assets.js
Web assets for alert sounds, can attach to text-feed or others.	
yahoo-smileys	web-assets.js
Web assets for smiley images, can attach to text-feed or others.	
text-chat	text-chat.js
Includes text-feed, text input area, and others for multiparty chat.	
text-feed-bubbles	text-chat-bubbles.js
Displays text messages as speech bubbles, contains items.	
text-item-bubbles	text-chat-bubbles.js
Displays a single text message, contained in feed.	
chat-images	web-assets.js
Web assets for optional border image of text-item-bubbles.	
text-feed-slack	text-chat-slack.js
Displays text chat feed similar to the popular Slack message app.	
text-item-slack	text-chat-slack.js
Displays a single chat item in the previous feed.	
text-date-item-slack	text-chat-slack.js
Displays a date/time item in the previous feed.	
text-overlay-item-slack	text-chat-slack.js
Displays overlay buttons for chat items in the previous feed.	
text-input-slack	text-chat-slack.js
Input controls for entering a text message, for the previous feed.	
text-overlay-file-slack	text-chat-slack.js
Displays a shared file overlay for chat items in the previous feed.	
user-roster	user-roster.js
Displays list of users and their attributes and controls.	
user-roster-data	user-roster.js
Data model for user-roster, using shared-storage.	
media-chat	media-chat.js
Multiparty conference user interface, includes text-chat, user-roster, flex-box and video-io.	
media-chat-data	media-chat.js
Data model for media-chat, using shared-storage.	
communicator-icons	web-assets.js
Web assets for icons to be used by media-chat, or others.	
toolbar-buttons.js	media-chat.js
Customizable toolbar buttons to be used by media-chat.	
overlay-menu	media-chat.js
Customizable overlay menu to be used by media-chat.	
volume-level	media-chat-js
Sounds activity or volume control to be used by media-chat.	
white-board	white-board.js
A white board app for drawing, texts and images.	
white-board-data	white-board.js
Data model for white-board, using shared-storage.	
locked-notepad	locked-notepad.js
A text editor for plain text, which can be locked.	
locked-notepad-data	locked-notepad.js
Data model for locked-notepad, using shared-storage.	
shared-editor	shared-editor.js
A rich text editor that can be shared and/or locked.	
shared-editor-data	shared-editor.js
Data model for shared-editor, using shared-storage.	
chess-board	chess-board.js
Displays a chess board, and allows playing without restriction.	
chess-board-data	chess-board.js
Data model for chess-board, using shared-storage.	
ludo-board	ludo-board.js
Displays a ludo board, and allows playing without restriction.	
ludo-board-data	ludo-board.js
Data model for ludo-board, using shared-storage.	
media-share	media-chat.js
Internally used to wrap a shared app within media-chat.	
slide-show	slide-show.js
Displays images, videos, or PDF documents with navigation.	
pdf-viewer	slide-show.js
Displays a PDF document using pdf.js, used by slide-show.	
phone-state	phone-call.js
State machine for phone register and call, using shared-storage.	
conference-state	phone-call.js
State machine for conference join, leave, and membership, using shared-storage.	
videos-control	phone-call.js
To start or stop media using video-io, attached to phone or conf.	
click-to-call	click-to-call.js
Displays a single clickable button, includes one or more phone or conf state, and supports a range of telephone use cases.	
native-electorn	native-electorn.js
Exposes native APIs such as raw socket connections, complex DNS query, frameless or native windows of the connected and locally running Native Electron app, to the web apps.	