5.2 Mobile/Web API Specification

Bright Pattern Documentation

Generated: 12/07/2021 3:48 pm

Table of Contents

Purpose

The Bright Pattern Contact Center *Mobile/Web API Specification* describes the methods, responses, and events of the Mobile API.

The Mobile API allows developers to integrate chat and voice interactions with mobile devices or third-party applications. This API can be used for development of rich contact applications, such as customer-facing mobile and web applications for advanced chat, voice, and video communications with Bright Pattern Contact Center-based contact centers.

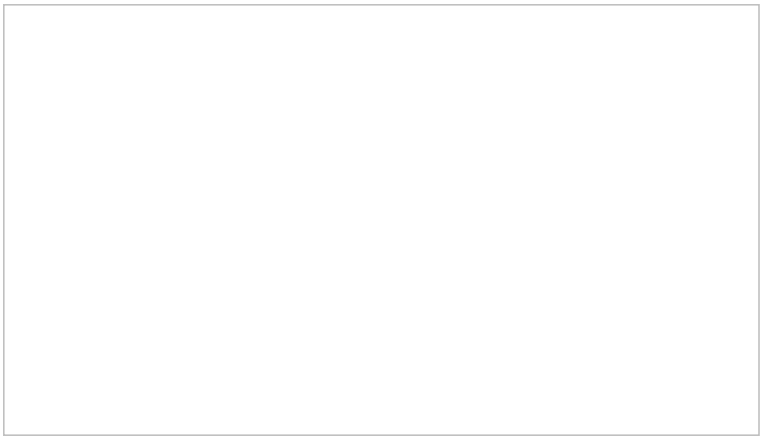
Audience

This guide is intended for the IT personnel and developers of custom applications for Bright Pattern Contact Center-based contact centers. Readers of this guide are expected to have expertise in web and mobile application development as well as a solid understanding of contact center operations and resources that are involved in such operations.

Sample Applications

Bright Pattern provides source code of sample customer-facing rich contact applications. This source code can be used as a reference for in-house application development or it can be embedded directly into the customer-facing mobile applications and/or web applications of your organization.

For web chat applications, you can download the source code by clicking the **Client application** button on the Messaging/CHat Scenario Entry page of the *Contact Center Administrator* application.



Clicking the Client Application button will download the source code for the web chat application

For mobile applications, you can obtain the source code by submitting a service request to Bright Pattern Customer Success Management.

General Information

The Bright Pattern Mobile/Web API is a RESTful API. The following is an example of an API request:

URL: http://localhost/clientweb/api/v1/chats?tenantUrl=example.com
Method: POST

Authorization: MOBILE-API-140-327-PLAIN appId="test", clientId="123"
Content-Type: application/json; charset=UTF-8
User-Agent: MobileClient

{"phone_number":"3001","parameters": {"first_name":"John","last_name":"Lee"}}

The TLS protocol can be used to ensure security of the data passed between Bright Pattern Contact Center and the customer-facing mobile/web applications. For secure communication, the above URL should be modified as follows:

https://localhost/clientweb/api/v1/chats?tenantUrl=example.com

Client Authentication Elements

The following elements are used for client authentication and must be specified in every client request:

Tenant URL

Tenant URL identifies your contact center. It corresponds to the domain name of your contact center that you see in the upper right corner of the Contact Center Administrator application after login.

appId

appId is the unique identifier of the Messaging/Chat scenario entry that will be used to associate your application with a specific scenario.

clientId

clientId is the unique identifier of the client application. It is used to identify communication sessions of a particular instance of the mobile application (i.e., of a specific mobile device). It must be generated by the mobile application in the UUID format. If clientId is set to *WebChat*, HTTP cookies will be used for client identification.

User-Agent

User-Agent specifies the type of client interface. For web applications, it is determined by the browser type. For mobile applications, this parameter shall always be set to *MobileClient*.

Additional Information

Standard HTTP response codes whose meaning conforms to the original specification (RFC 2616) are not discussed in this document. For specification of such responses, see section 10 of http://www.ietf.org/rfc/rfc2616.txt. This document only specifies the response codes whose description deviates from the original specification (e.g., is defined more narrowly or has a different meaning).

Note that a client application can only have one active communication session at a time.

Check Availability

Returns the current status of configured services.

Request

URI

http[s]://<host>/clientweb/api/v1/availability?tenantUrl=<tenantUrl>

HTTP Method

GET

Response

Content type

application/json

Example Response

{
"chat": "<available|notAvailable>"
}

Status Meanings

Status Name	Description
available	The requested service is available.
notAvailable	The office providing the requested service is currently closed.

Codes

Code	Description
200	Names of configured services with current status
401	Unauthorized

Expected Parameters

This returns an array of expected parameters. The parameters are optional. They may be used, for example, to look for the most qualified agents to handle the requested chat sessions and/or to display user information for the agents. These parameters are initially specified for the Messaging/Chat scenario entry through which this mobile application will communicate with the contact center.

Currently, the following parameter types are supported:

- PHONE
- TEXT
- EMAIL
- FIRST_NAME
- LAST_NAME
- ACCOUNT
- COMPANY

URL

http[s]://<host>:<port>/clientweb/api/v1/parameters?tenantUrl=<tenantUrl>

Method

GET

Returns

200 – Names and types of parameters that may be expected from the client application (content type: application/json)

Example

```
{
"parameters":[
        {
        "<name>":"<type>"
        },
        ...
]
}
```

Errors

401 – Unauthorized

< Previous | Next >

Request Chat

Request Chat initiates a chat session. It provides values of all or some of the expected parameters, and it may also contain the phone number of the mobile device. Depending on the scenario entry configuration, a callback can be initiated automatically to the specified number as soon as the chat request is delivered to the agent.

Note that if the mobile/web scenario entry is not configured for automatic callback, the agent can still use this number to call the mobile user manually, either upon the agent's own initiative or when asked to do this via a chat message from the mobile user.

Parameter *from* may be used to specify either the device owner's name or phone number. It will be placed in the scenario variable *$(item.from)*.

URL

http[s]://<host>:<port>/clientweb/api/v1/chats?tenantUrl=<tenantUrl>

Method

POST

Request body

Names and values of record fields (content type: application/json)

Example

```
{
        “phone_number”:”phone number for callback, if necessary”,
        “from”:”propagated into scenario variable $(item.from)”,
        “parameters”:
        {  //list of expected parameters
        "parameter_1":"value_1",
        …,
        "parameter_n":"value_n"
        },
}
```

Returns

200 – Chat session properties (content type: application/json)

Example

```
{
"chat_id":”<chat ID>”,
        “state”: “<queued|connected|ivr>”,
        “ewt”:<estimated wait time for a chat in state queued>,
        “is_new_chat”: true|false,
        “phone_number”:”phone number that will be used for callback”
}
```

Errors

400 – Bad request

```
{
"error_code": <error code>,
“error_message”:”<error message>”
}
```

401 – Unauthorized

< Previous | Next >

Get Active Chat

Get Active Chat returns the properties of an active chat session.

URL

http[s]://<host>:<port>/clientweb/api/v1/chats/active?tenantUrl=<tenantUrl>

Method

GET

Returns

200 – Chat session properties (content type: application/json)

Example

```
{
"chat_id":"<chat ID>",
        "state":"<queued|connected|ivr>",
        "ewt":<estimated wait time for a chat in state queued>,
        "phone_number":"<phone number that will be used for callback>"
}
```

Errors

401 – Unauthorized

404 – Not found

< Previous | Next >

Send Events

Send Events sends events from the client to the server side and returns events from the server.

URL

http[s]://<host>:<port>/clientweb/api/v1/chats/{chatId}/events?tenantUrl=<tenantUrl>

Method

POST

Request Body

Array of chat events (content type: application/json)

Example

```
{
    "events":[<array of chat events>]
}
```

Returns

200 – Array of new chat events from the server side (content type: application/json)

Example

```
{
"events":[<array of new chat events from server>]
}
```

Errors

401 – Unauthorized

404 – Not found

< Previous | Next >

Get Chat History

Get Chat History returns all client events and all server events for the current session. Event parameter *timestamp* can be used to restore the correct message order.

URL

http[s]://<host>:<port>/clientweb/api/v1/chats/{chatId}/history?tenantUrl=<tenantUrl>

Method

GET

Returns

200 – Array of chat events (content type: application/json)

Example

```
{

"events":[<array of chat events>]

}
```

Errors

401 – Unauthorized

404 – Not found

< Previous | Next >

Get New Chat Events

Get New Chat Events returns new chat events for the current session. If there are no new events, the server can keep the request active during for about 5–15 seconds. If the client sends another request to get new events while the previous request is kept active, the server will respond with error 400 (Bad request).

URL

http[s]://<host>:<port>/clientweb/api/v1/chats/{chatId}/events?tenantUrl=<tenantUrl>

Method

GET

Returns

200 – Array of chat events (content type: application/json)

Example

```
{
"events":[<array of chat events>]
}
```

Errors

400 – Bad request

401 – Unauthorized

404 – Not found

< Previous | Next >

Upload File

Upload File uploads a file to the server. The maximum file size is 25 MB. The server will store up to 1000 files. Newly received files will replace the oldest files. This method can be used, for example, to send pictures from the client mobile application to the contact center agents.

URL

http[s]://<host>:<port>/clientweb/api/v1/files?tenantUrl=<tenantUrl>

Method

POST

Request Body

File to be uploaded (content type: mulitpart/form-data)

Example

```
file={file to upload}
```

Returns

200 – file ID to be used in the *chat_session_file* message (content type: application/json)

Example

```
{
"file_id":"file ID"
}
```

Errors

401 – Unauthorized

< Previous | Next >

Get File

Get File downloads a file from the server using the received file ID. See server event Chat Session File.

URI

http[s]://{host:port}/clientweb/api/v1/chats/{chatId}/files/{fileId}

HTTP Method

GET

Parameters

- **fileId** - File ID to download
- **chatId** - Current chat ID

Request Body

ID of the file to be downloaded

Example

fileId={file id of file to download}

chatId=current chat id

Returns

200 – File content type

Errors

401 – Unauthorized

404 – Not Found

< Previous | Next >

Get Agent Profile Photo

Get Agent Profile Photo downloads a profile photo of the specified agent.

URL

http[s]://{host:port}/clientweb/api/v1/chats/{chatId}/profilephotos/{partyId}

Method

GET

Request Body

The request body is the party ID of the agent participating in the current chat session.

Note: The party ID is provided with the chat event (polled at *https://{host:port}/clientweb/api/v1/chats/{chatId}/events?tenantUrl={tenantUrl}*). For example, you can see *party_ID* in the event message for chat_session_party_joined.

Example

partyId={party id of agent for current chat session}

chatId=current chat id

Returns

200 – File content type

Errors

401 – Unauthorized

404 – Not found

< Previous | Next >

Client Events

Chat Session Message

Chat Session Message contains a new chat message.

The *msg_id* parameter is a unique identifier of the message in the following format*(chatId:messageNumber).*

The *chatId* is a unique identifier of the chat session that is returned by the server in response to client's request for chat.

The *messageNumber* is the ordinal number of the given message in the chat exchange.

Format

```
{
"event": "chat_session_message",
"msg_id": "<message id>",
"msg": "<chat message>"
}
```

Chat Session Typing

Chat Session Typing is sent when the user of the client application starts typing.

Format

```
{
"event":
"chat_session_typing"
}
```

Chat Session Not Typing

Chat Session Not Typing is sent when the user of the client application stops typing.

Format

```
{
"event": "chat_session_not_typing"
}
```

Chat Session Form Data

Chat Session Form Data contains the data requested by the server via event Chat Session Form Show.

Format

```
{
"event": "chat_session_form_data",
"form_request_id": "<form request ID from the corresponding server event>",
"form_name": "form name",
"data": {
"param1": "value1",
...
}
}
```

Chat Session End

Chat Session End ends the chat session.

Format

```
{
"event": "chat_session_end"
}
```

Chat Session Disconnect

This ends the current chat conversation but keeps the session open (i.e., controlled by the scenario). This message can be used instead of Chat Session End when additional information may be expected from the server (e.g., a survey form).

Format

```
{
"event": "chat_session_disconnect"
}
```

Chat Session Signaling

This is a container message for WebRTC signaling messages from client to server. These messages are not stored in the chat session history.

Format

```
{
"event": "chat_session_signaling",
"msg_id": "<message id>",
"destination_party_id": "<agent party id>",
"data": "<signaling message>"
}
```

Chat Session File

Chat Session File provides notification that the client has uploaded a file to the server. Parameter *file_id* identifies the uploaded file.

Format

```
{
"event": "chat_session_file",
"msg_id": "<message id>",
"file_type": "<image|attachment>",
"file_id": "<file id>",
}
```

< Previous | Next >

Server Events

Chat Session Status

Chat Session Status updates the current state of the chat session. If the state is *failed*, the client application shall assume that the chat session no longer exists.

Format

```
{
"event": "chat_session_status",
"state": "queued|connecting|connected|failed|completed",
"ewt": "estimated waiting time for queued status"
}
```

Chat Session Ended

Chat Session Ended indicates a normal termination of the chat session (e.g., when the chat session is closed by the agent). The client application shall assume that the chat session no longer exists.

Format

```
{
"event": "chat_session_ended"
}
```

Chat Session Party Joined

Chat Session Party Joined indicates that a new party (a new agent) has joined the chat session. The*party_id* is a unique identifier of the chat session on the server side. A separate identifier is generated for each party (agent) that handles this chat session.

Format

```
{
"event": "chat_session_party_joined",
"party_id": "<party id>",
"first_name": "<party first name>",
"last_name": "<party last name>",
"display_name": "<party display name>",
"type": "<scenario|external|internal>",
"timestamp": "<UTC timestamp in milliseconds>"
}
```

Chat Session Party Left

Chat Session Party Left indicates that one of the existing parties (an agent) has left the chat session. The*party_id* is a unique identifier of the chat session on the server side. A separate identifier is generated for each party (agent) that handles this chat session.

Format

```
{
"event": "chat_session_party_left",
"party_id": "<party id>",
"timestamp": "<UTC timestamp in milliseconds >"
}
```

Chat Session Message

Chat Session Message contains a new chat message.

The *msg_id* parameter is a unique identifier of the message.

The *partyId* is a unique identifier of the party that sends this message.

A separate identifier is generated for each party (agent) that handles this chat session.

Format

```
{
"event": "chat_session_message",
"party_id": "<party id>",
"msg_id": "<message id>",
"msg": "<chat message>",
"timestamp": "<UTC timestamp in milliseconds >"
}
```

Chat Session Typing

Chat Session Typing is sent when the agent starts typing. The *party_id* is a unique identifier of the chat session on the server side. A separate identifier is generated for each party (agent) that handles this chat session.

Format

```
{
"event": "chat_session_typing",
"party_id": "<party id>"
}
```

Chat Session Not Typing

Chat Session Not Typing is sent when the agent stops typing. The *party_id* is a unique identifier of the chat session on the server side. A separate identifier is generated for each party (agent) that handles this chat session.

Format

```
{
"event": "chat_session_not_typing",
"party_id": "<party id>"
}
```

Chat Session Form Show

Chat Session Form Show is sent to the request user's input from the client application to be entered via a predefined form. The client application is normally supposed to display the specified form to the user and send back the entered data via event Chat Session Form Data.

Format

```
{
"event": "chat_session_form_show",
"form_request_id": "<request ID that will be used by the client application to associate the response >",
"form_name": "<predefined form name known to the client application>"
"form_timeout": "form timeout"
}
```

Chat Session Timeout Warning

Chat Session Timeout Warning is sent to request that the specified text be displayed to the user of the client application. Typically, it is used to display an inactivity warning message.

Format

```
{
"event": "chat_session_timeout_warning",
"msg": "<warning text>",
"timestamp": "<UTC timestamp in milliseconds>"
}
```

Chat Session Inactivity Timeout

Chat Session Inactivity Timeout indicates the termination of the chat session due to the user's inactivity and provides text to be displayed to the user. The client application shall assume that the chat session no longer exists.

Format

```
{
"event": "chat_session_inactvity_timeout",
"msg": "<inactivity timeout text>",
"timestamp": "<UTC timestamp in milliseconds>"
}
```

Chat Session Signaling

This is a container message for WebRTC signaling messages from server to client. These messages are not stored in the chat session history.

Format

```
{
"event": "chat_session_signaling",
"msg_id": "<message id>",
"party_id": "<party_id>",
"data": "<signaling message>"
}
```

Chat Session File

Chat Session File provides notification that a file is being sent to the client--the client can use *file_id* to download the file.

The *msg_id* parameter is a unique identifier of the corresponding message.

The *partyId* is a unique identifier of the party that sends this message.

Format

```
{
“event”: “chat_session_file”,
“party_id”: “<party id>”,
“msg_id”: “<message id>”,
“file_id”: “<file id>”,
“file_type”: “<image|attachment>”,
“timestamp”: “<event time in Unix format>”
}
```

< Previous | Next >

Signaling Message Types

Request Call

This message is sent from the client to request a WebRTC call.

Format

```
{
“type”: “REQUEST_CALL”,
“offerVideo”: true|false
}
```

Call Rejected

This message is sent from the server to indicate that a requested call was rejected.

Format

{

“type”: “CALL_REJECTED”,

“reason_code”: <reject reason code>,

“reason”: TBD

}

Offer Call

This message is sent from the server to start a WebRTC session.

Format

{

“type”: “OFFER_CALL”,

“offerVideo”: true|false,

“sdp”: “<session description string>”

}

Answer Call

This message is sent from the client to confirm the WebRTC session started by the server.

Format

{

“type”: “ANSWER_CALL”,

“sdp”: “<session description string>”

}

ICE Candidate

This message is sent from the client or server with ICE candidate description.

Format

```
{
“type”: “ICE_CANDIDATE”,
“sdpMid”: “<candidate id>”,
“sdpMLineIndex”: “<candidate index>”,
“candidate”: “<candidate description>”
}
```

End Call

This message is sent from the client or server to end the current WebRTC session.

Format

```
{
“type”: “END_CALL”
}
```

< Previous