Chat API

Published: 2019-02-18 12:17

The boost.ai Chat API is your toolbox for building virtual assistant (VA) front ends. Start here to learn about the core components available for you to build with, and to get some guidance on how to start thinking about designing your interface to the platform.

Getting started

The easiest way to get started with the API is to use the example code. You don’t need to install anything; you can just open it in a terminal window or in a programming language of your choice. Let’s get started!

Request URI

Calls to the API use the endpoint:

```
POST /api/chat/v2 HTTP/1.1
```

Commands

Interaction with the the Chat API revolves around a conversation. You interact with a conversation using commands. The commands most commonly used are:

COMMAND	DESCRIPTION
START	Start conversation
POST	Post data

Additionally, you can download the history of a conversation, delete it and more, using the following commands:

COMMAND	DESCRIPTION
DOWNLOAD	Download conversation
RESUME	Resume conversation
DELETE	Delete conversation
FEEDBACK	Conversation feedback
TYPING	End user is typing
POLL	Poll human chat
POLLSTOP	Stop human chat mode
STOP	Stop conversation
LOGINEVENT	Login event

Posting data

START: Starting a conversation

You initiate a conversation by calling the START command.

The simplest form of the command is with a single parameter:

```
{ "command": "START" }
```

Additional parameters can be sent with the START command:

```
{
  "command": "START",
  "language": String,            // (optional) Language
  "filter_values": Array,        // (optional) Array of filter value strings
  "trigger_action": Number,      // (optional) Action to trigger
  "auth_trigger_action": Number, // (optional) Action to trigger
  "user_token": String,          // (optional) User token
  "custom_payload": Any,         // (optional) Custom fields
  "client_timezone": String,     // (optional) Timezone
}
```

- `language` is a BCP47 code string. Examples: ‘en-US’, ‘fr-FR’ and ‘sv-SE’. This will cause the VA to respond in the specified language.
- `filter_values` is a list of strings, e.g. `['login', 'production']`. Filter values are used to filter actions in the action flow.
- `user_token` identifies an authenticated user.
- `trigger_action` is a specific action id you want to trigger instead of the welcome message configured in Settings -> System action Triggers. If you have enabled GDPR this parameter will return an error message.
- `auth_trigger_action` is a specific action id you want to trigger for authenticated users instead of the welcome message configured in Settings -> System action Triggers. If you have enabled GDPR this parameter will return an error message.
- `custom_payload` is forwarded to the API Connector and External API’s. You can set this parameter to any JSON value.
- `client_timezone` is forwarded to the API Connector and External API’s. This parameter can tell an API which timezone the client is currently in. The format is listed here . Use the column **TZ database name** in the list.

Example: Start a conversation.

```
$ sudo apt install jq curl
$ curl -sS -H "Content-Type: application/json" \
     -d '{"command": "START"}' \
     -X POST https://<servername_or_ipaddress>/api/chat/v2
```

POST: Posting data

When you have obtained a conversation_id, you can post data to the server:

```
{
  "command": "POST",
  "conversation_id": String,     // Conversation ID
  "type": String,                // Element type
```

```
  "filter_values": Array,        // (optional) Array of filter values
  "user_token": String,          // (optional) User token
  "clean": Boolean,              // (optional) Clean text
  "context_intent_id": Number,   // (optional) Context intent id
  "id": String,                  // (optional) Element id
  "value": Any,                  // (optional) Data value(s)
  "custom_payload": Any,         // (optional) Custom fields
  "client_timezone": String,     // (optional) Timezone
}
```

Fields

FIELD	TYPE	REQUIRED	DESCRIPTION
`command`	String	✓	Command to execute
`conversation_id`	String	✓	Identifies the conversation. The conversation id is generated by the API.You are responsible for picking up the conversation id from a response, and keeping it as a state variable in your client.
`type`	String	✓	The type of request.
`clean`	Boolean		Return clean text.
`filter_values`	Array		Array of filter value strings. These values can be used in filtering data in the action flow.
`user_token`	String		Identifies an authenticated user.
`context_intent_id`	Number		Intent id to set conversation in specific context.
`id`	String		The id of a button or a bot question.
`value`	Any		The value of the request.
`custom_payload`	Any		An object which is forwarded to External API's.
`client_timezone`	String		Forwarded to the API Connector and External API's. This parameter can tell the API which timezone the client is currently in. The format is listed here . Use the column **TZ database name** from the list.

Clean

If `clean=true`, the API will return clean text instead of HTML in all responses.

Types

TYPE	RESULT OF	REQUIRED FIELDS
"text"	A chat message	
"trigger_action"	Manually triggered action	id
"action_link"	An action link (button)	id

“external_link”	An external link	id
“feedback”	A like or dislike button	id, value

Message text

Use the **text** type when sending chat messages to the server:

```
{
  "conversation_id": String,
  "command": "POST",
  "type": "text",
  "value": String
}
```

> Note with regards to Human Chat pollingWhen you post message text, the response from the server will contain a `posted_id` property. Use this property in your “POLL” commands to check for new messages.

Message feedback

User feedback allows your users to give thumbs up/down responses in your chat panel. Use the *feedback* type when sending message feedback:

```
{
  "conversation_id": String,
  "command": "POST",
  "type": "feedback",
  "id": String,                    // Message ID
  "value": String                  // Feedback value (see below)
}
```

The `id` attribute should correspond to an id in a response.

Feedback values

VALUE	DESCRIPTION
“positive”	Result of a like click
“remove-positive”	Result of a like unclick
“negative”	Result of a dislike click
“remove-negative”	Result of a dislike unclick

Action link

If a response message contains a message_id, you can provide feedback to the API in the form:

```
{
  "conversation_id": String,
  "command": "POST",
  "type": "action_link",
  "id": String
}
```

External link

External links do not trigger a response from the server, but they should be sent for logging purposes.

```
{
  "conversation_id": String,
  "command": "POST",
  "type": "external_link",
  "id": String
}
```

Trigger action

Use this request type to trigger action flow elements directly.

```
{
  "conversation_id": String,
  "command": "POST",
  "type": "trigger_action",
  "id": String                // The id of the action flow item
}
```

RESUME: Resuming a conversation

Use the RESUME command to list previous responses in a given conversation. When you resume a conversation, you supply a conversation id:

```
{
  "command": "RESUME",
  "conversation_id": String,
}
```

`conversation_id` uniquely identifies a conversation. If you supply a conversation_id, the chat history of the associated conversation is returned.

DELETE: Deleting a conversation

To delete a conversation, post a delete command:

```
{
  "command": "DELETE",
  "conversation_id": String
}
```

When you delete a conversation, the API will delete or overwrite:

- Message texts sent to and from the API
- Middle layer data sent through web hooks

POLL: Poll for data

When the API returns `poll=true`, the conversation is in Human Chat mode.

You can poll the server for more conversation data using the last known response id. Responses will contain any response with an id larger than the supplied id. The maximum number of responses that can be returned is 100.

```
{
  "command": "POLL",
  "conversation_id": String,
  "value": String             // Last known Response ID.
}
```

POLLSTOP: Stop Human Chat

POLLSTOP will stop the Human Chat state and return the conversation to AI mode.

```
{
  "command": "POLLSTOP",
  "conversation_id": String
}
```

TYPING: User activity

When the user is typing, you might want to send a `typing` message to the server. If the conversation router is in Human Chat mode, this will update the chat status in the Human Chat.

```
{
  "command": "TYPING",
  "conversation_id": String
}
```

Good practice would be to send this message with a 5 second delay timer:

```
<!DOCTYPE html>
<html>
<head><title>My awesome Chat Client</title></head>
<body>
  <textarea id="chat_textarea"></textarea>
  <script>
    <!-- Javascript here -->
  </script>
</body>
</html>
```

Paste the following code into the script tag above.

```
(() => {
  document
    .getElementById('chat_textarea')
    .addEventListener('keypress', userIsTyping);
  function userIsTyping() {
    window.clearTimeout(window.isTypingTimer);
    window.isTypingTimer = window.setTimeout(function() {
      fetch('https://<servername_or_ipaddress>/api/chat/v2', {
          method: 'POST',
          headers: { 'Content-Type': 'application/json' },
          body: JSON.stringify({
            command: "TYPING",
            conversation_id: "<conversation_id>"
          })
        }).then(function(res) {
            res.json().then(function(data) {
              console.log(data)
            })
        }).catch(function(err){
          console.error('Something went wrong when posting data', err);
        });
    }, 5000);
```

```
  }
})();
```

FEEDBACK: Conversation feedback

When a conversation ends, you might want to give the user the opportunity to give feedback on the conversation.

```
{
  "command": "FEEDBACK",
  "conversation_id": String,
  "value": {
    "rating": Integer,            // 1 or 0
    "text": String                // Feedback text
  }
}
```

The rating parameter should be the integer 1 or 0. If rating is missing, its value will be 0.

If you want your users to provide feedback in the form of text in addition to rating, use the `text` attribute.

DOWNLOAD: Download conversation history

Use this command to download a text version of a conversation.

> Tip: you can also use a GET call to download a conversation (see Appendix).

Either conversation*id or user*token should be present in payload. Download by conversation_id:

```
{
  "command": "DOWNLOAD",
  "conversation_id": String
}
```

Download by user_token:

```
{
  "command": "DOWNLOAD",
  "user_token": String
}
```

Returns a HTTP response with mime-type “text/plain;charset=utf-8”.

Example usage with Filesaver :

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>My awesome Chat Client</title>
</head>
<style type="text/css">
    * { font-size: 1.2em; }
</style>
<body>
  Conversation ID: <input type="text" id="conv-id" style="width:100%;" />
  <br/><br/>
  <button type="button" id="download-button">Download conversation</button>
  <script>
    const getDownload = () => {
      const conversation_id = document.getElementById('conv-id').value
```

```
      fetch('https://<servername_or_ipaddress>/api/chat/v2', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({
          command: "DOWNLOAD",
          conversation_id: conversation_id
        })
      }).then(function(res) {
        res.text().then(function(data) {
          let blob = new Blob([data], {type: "text/plain;charset=utf-8"});
          saveAs(blob, "conversation.txt" );
        })
      }).catch((err) => {
        console.error(err);
      });
    };

    (() => {
      document
        .getElementById('download-button')
        .addEventListener('click', getDownload);
    })();
  </script>
  <script src="https://tinyurl.com/y9b7z4tl"></script>
</body>
</html>
```

STOP: Stopping a conversation

The STOP command will block the conversation. This is useful in authenticated conversation flows when you want to force a new conversation to be created with the START command.

```
{
  "command": "STOP",
  "conversation_id": String
}
```

LOGINEVENT: Login event

When a response requires authentication, and the conversation is not authenticated, the chat cannot continue unless the user authenticates. When authentication has been accomplished, a LOGINEVENT command will resume the dialog and give the expected response.

```
{
  "command": "LOGINEVENT",
  "user_token": String
}
```

Receiving data

When receiving data, you'll receive a list of responses, which should indicate the behavior of your client. This content is distinguished with a type parameter. The API supports a list of predefined response types.

A response consists of a conversation_id, a conversation state, and a list of elements.

```
{
  "conversation": {
    "id:" String,                       // Conversation ID
    "state": {
      "is_blocked": Boolean,            // Conversation is blocked for posts
      "poll": Boolean,                  // Client should poll
      "max_input_chars": Integer,       // Maximum characters allowed in a request
text
      "unauth_conversation_id": String  // Unauthorized conversaton id
    }
  },
  "response": {
    "id": String,                       // Response ID
    "source": String,                   // Message source. "bot" or "human"
    "language": String,                 // bcp-47 language code
    "elements": [                       // List of response objects
      {
        "type": String,                 // Response type (e.g. "links", "text")
        "payload": [                    // List of elements
          {
            "type": String,             // (optional) The type of element
            ...
          },
          ...
      ]
    ]
  }
}
```

Example response:

```
{
  "conversation": {
    "id": "<conversation_id>",
    "state": {
      "max_input_chars": 110,
      "is_blocked": true
    }
  },
  "response": {
    "id": "120303",
    "source": "bot",
    "language": "en-US",
    "elements": [
        {
          "type": "text",
          "payload": { "text": "Hi!" }
        },{
          "type": "text",
          "payload": { "text": "My name is James, how can I help you?" }
        },{
          "type": "text",
          "payload": { "text": "Write to me in any language." }
        }
    ]
  }
}
```

Attributes

ATTRIBUTE	TYPE	REQUIRED	DESCRIPTION
`conversation`	Object	✓	Conversation object.
`conversation.id`	String	✓	Identifies the conversation.
`conversation.state`	Object	✓	Conversation state object.
`conversation.state.is_blocked`	Boolean		When `true`, the conversation is blocked.
`conversation.state.authenticated_user_id`	String		Identifier for the end user, if authenticated.
`conversation.state.unauth_conversation_id`	String		Conversation id used before the user was authenticated.
`conversation.state.privacy_policy_url`	String		Privacy policy URL.
`conversation.state.allow_delete_conversation`	Boolean		If `true`, the DELETE command is operational
`conversation.state.poll`	Boolean		Whether the conversation is in Human Chat state. The client should POLL the server for more data.
`conversation.state.human_is_typing`	Boolean		`true` if human is typing in Human Chat.
`conversation.state.max_input_chars`	Integer		Maximum characters allowed in a text POST.
`conversation.state.skill`	String		A string containing the skill set on the predicted intent.
`response`	Object		Response from an interactive conversation
`response.id`	String	✓	The id of the response.
`response.source`	String	✓	The source of the response. Either “bot” or “client”.
`response.language`	String	✓	Detected language bcp-47 code.
`response.elements`	Array	✓	A list of response elements.
`response.elements[].type`	String	✓	The data type of the response.
`response.elements[].payload`	Object	✓	Element data.
`responses`	Array		List of historic `response` objects.

“POLL”, “RESUME”

Poll and resume requests will return the following attributes in response objects:

ATTRIBUTE	TYPE	REQUIRED	DESCRIPTION
`response.feedback`	String		Message feedback.
`response.source_url`	String		Server URL used by the chat client.
`response.link_text`	String		The text of a link that was clicked by an end user.
`response.avatar_url`	String		Avatar URL if message is from Human Chat.

Example response

```
{
  "conversation": {
    "id": "<conversation_id>",
    "state": { "max_input_chars": 110 }
  },
  "response": {
    "source": "bot",
    "language": "en-US",
    "id": "11031",
    "avatar_url": "https://master.boost.ai/img/james.png",
    "elements": [
      {
        "type": "links",
        "payload": {
          "links": [
            {
              "id": "1234567891",
              "type": "external_link",
              "text": "Click 1",
              "url": "http://noop.com/index1.html"
            },
            {
              "id": "1234567892",
              "type": "external_link",
              "text": "Click 2",
              "url": "http://noop.com/index2.html"
            },
            {
              "id": "1234567893",
              "type": "external_link",
              "text": "Click 3",
              "url": "http://noop.com/index3.html"
            }
          ]
        }
      }
    ]
  }
}
```

Response types

When receiving data, the `type` parameter indicates how your client should render their data. The following are the supported element types in the API.

TYPE	DESCRIPTION
text	Plain text response.

html	HTML content.
image	Image with URL.
video	Youtube, Vimeo or Wistia video with URL.
google_directions	Google map with directions.
google_places	Google map with place identifier.
google_location	Google location with place identifier.

HTML message

An HTML string you can render as a chat bubble in your client.

Example:

```
{
  "conversation": {
    "id": "<conversation_id>",
    "state": { "max_input_chars": 110 }
  },
  "response": {
    "id": "14001",
    "source": "bot",
    "language": "en-US",
    "elements": [
      {
        "type": "html",
        "payload": {
          "html": "<p>Hello I am your friend.</p>"
        }
      },
      {
        "type": "html",
        "payload": {
          "html": "<p>What can I help you with?</p>"
        }
      }
    ]
  }
}
```

Text message

A text string you can render as a chat bubble in your client.

To trigger pure text responses, send `clean = true` in your post message. This applies to all requests, including POST, RESUME, and DOWNLOAD:

```
{
  "conversation_id": String,
  "command": "POST",
  "type": "text",
  "clean": true,                  // Instruct the virtual assistant to return clean
text
  "value": String                 // The chat message
}
```

Example:

```
{
```

```
  "conversation": {
    "id": "<conversation_id>",
    "state": { "max_input_chars": 110 }
  },
  "response": {
    "id": "14007",
    "source": "bot",
    "language": "en-US",
    "elements": [
      {
        "type": "text",
        "payload": {
          "text": "Hello I am your friend."
        }
      },
      {
        "type": "text",
        "payload": {
          "text": ">What can I help you with?"
        }
      }
    ]
  }
}
```

Link list

A list of links.

```
{
  "conversation": {
    "id": String,
    "state": Object
  },
  "response": {
    "source": "bot",
    "language": String,
    "id": String,
    "elements": [
      {
        "type": "links",
        "payload": {
          "links": [
            {
              "id": String,
              "type": String,       // "action_link" or "external_link".
              "text": String,
              "function": String    // "APPROVE" or "DENY" if present.
            },
            ...
          ]
        }
      }
    ]
  }
}
```

Image

An image.

```
{
  "conversation": {
    "id": String,
```

```
    "state": Object
  },
  "response": {
    "source": "bot",
    "language": String,
    "id": String,
    "elements": [
      {
        "type": "image",
        "payload": {
          "url": String
        }
      }
    ]
  }
}
```

Video

A video.

```
{
  "conversation": {
    "id": String,
    "state": Object
  },
  "response": {
    "source": "bot",
    "language": String,
    "id": String,
    "elements": [
      {
        "type": "video",
        "payload": {
          "source": String  // One of "youtube", "vimeo" or "wistia"
          "url": String,
          "fullscreen": Boolean
        }
      }
    ]
  }
}
```

Google Directions

Directions from one location to another using the Google Directions API.

```
{
  "conversation": {
    "id": String,
    "state": Object
  },
  "response": {
    "source": "bot",
    "language": String,
    "id": String,
    "elements": [
      {
        "type": "google_directions",
        "payload": {
            "GD_START_LAT": String,
            "GD_START_LNG": String,
            "GD_END_LAT": String,
            "GD_END_LNG": String,
```

```
            "GD_ENCODED_PATH": String,
            "START_ADDRESS": String,
            "END_ADDRESS": String
          }
        }
      }
    ]
  }
}
```

Google Places

Google Map coordinates for the given location, including a marker for a given object type, using the Google Places API.

```
{
  "conversation": {
    "id": String,
    "state": Object
  },
  "response": {
    "source": "bot",
    "language": String,
    "id": String,
    "elements": [
      {
        "type": "google_places",
        "payload": {
            "GP_TITLE": String,
            "GP_LATITUDE": String,
            "GP_LONGITUDE": String,
            "GP_NW_LAT": String,
            "GP_NW_LNG": String,
            "GP_SW_LAT": String,
            "GP_SW_LNG": String,
            "PLACE": String
          }
        }
      }
    ]
  }
}
```

Google Locations

Google map for the given location using the Google Location API.

```
{
  "conversation": {
    "id": String,
    "state": Object
  },
  "response": {
    "source": "bot",
    "language": String,
    "id": String,
    "elements": [
      {
        "type": "google_location",
        "payload": {
            "GL_FORMATTED_ADDRESS": String,
            "GL_LATITUDE": String,
            "GL_LONGITUDE": String,
            "GL_NW_LAT": String,
```

```
            "GL_NW_LNG": String,
            "GL_SW_LAT": String,
            "GL_SW_LNG": String
          }
        }
      }
    ]
  }
}
```

HTTP status codes

If successful, the API returns HTTP status 200 (OK). Known server errors are returned with code 400, usually with a description of the error message.

CODE	DESCRIPTION	COMMENT
200	OK	
400	User error	Occurs when there is an error in the posted data or in the configuration of an action flow.
403	Unauthorized	Occurs when you send “POST” commands in a blocked conversation or an authenticated conversation has expired.
500	Server error	Something is wrong on the server.

Errors are accompanied by an error object in the response:

```
{
  "error": String          // Error message
}
```

Security

Boost.ai's cloud endpoints use HTTPS for transport security. In addition, you can hash your data to validate the integrity and origin of the payload. Request hashing is enabled in the Admin Panel. When request hashing is enabled, you will:

- Obtain a key: Boost.ai's standard method of sending the key is a password-protected zipfile sent by email. The password will then be sent by SMS.
- Decode your key: The key you receive is a Base64 encoded string. To use the key you must first decode it.
- Hash your request: When your key is decoded, you can use it to hash your request data.
- Send your request: The hash is sent using the `X-Hub-Signature` HTTP header.

Hash example - Python

```python
#!/usr/bin/python3
import hashlib
import hmac
import base64
import requests
import json

# Parameters
url = "https://<servername_or_ipaddress>/api/chat/v2"
request_data = {"command": "START"}
encoded_key = "<your_key>"

# Hash and send
key = base64.b64decode(encoded_key)
request_data_encoded = json.dumps(request_data).encode('utf-8')
hmac_ = hmac.new(key, request_data_encoded, digestmod=hashlib.sha512)
digest = hmac_.hexdigest()
headers = {"content-type": "application/json", "X-Hub-Signature": str(digest)}
request = requests.post(url, json=request_data, headers=headers, timeout=(5))
print("Response: " + str(request.text))
```

Hash example - Javascript

Using CryptoJS: https://code.google.com/archive/p/crypto-js .

```javascript
const url = "https://<servername_or_ipaddress>/api/chat/v2";
const request_data = { "command": "POST" };
const encoded_key = "<your_key>";

// Hash and send
const key = CryptoJS.enc.Base64.parse(encoded_key);
const request_data_encoded = JSON.stringify(request_data);
const hmac_ = CryptoJS.HmacSHA512(request_data_encoded, key);
const signature = hmac_.toString(CryptoJS.enc.Hex);

fetch('https://<servername_or_ipaddress>/api/chat/v2', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json', 'X-Hub-Signature': signature }
    body: JSON.stringify(request_data)
  }).then(res => {
    res.json().then(data => {
      console.log(data); // Process the response data
    })
  })
});
```

Examples

The following examples show how you can start a new conversation in different programming languages.

Python

```
import requests
request = requests.post("https://<servername_or_ipaddress>/api/chat/v2",
    json={"command": "START"},
    headers={"Content-Type": "application/json"})
print("Response: " + str(request.text))
```

Javascript

```
fetch('https://<servername_or_ipaddress>/api/chat/v2', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' }
    body: JSON.stringify({ command: "START" })
  }).then(res => {
    res.json().then(data => { console.log(data); })
  });
```

Node.js

```
const req = require('http').request({
    hostname: '<servername_or_ipaddress>',
    path: '/api/chat/v2',
    method: 'POST',
    headers: { 'Content-Type': 'application/json' }
}, res => {
    let data = '';
    res.on('data', chunk => { data += chunk.toString(); });
    res.on('end', () => { console.log(JSON.parse(data)); });
});
req.write(JSON.stringify({ command: 'START' }));
req.end();
```

Linux terminal

1. Install **jq** and **curl**

```
$ sudo apt install jq curl
```

1. Start conversation

```
$ curl -sS -H "Content-Type: application/json" \
       -d '{"command": "START"}' \
       -X POST https://<servername_or_ipaddress>/api/chat/v2
```

Appendix - GET endpoints

Download conversation

Use this command to download a text version of a conversation. Either conversation*id or user*token should be present in the requested URL.

Download by conversation_id:

```
GET /api/chat/v2/conversation/download/<conversation_id> HTTP/1.1
```

Download by user_token:

```
GET /api/chat/v2/conversation/download?user_token=someTokenValue HTTP/1.1
```

Returns a human readable conversation history for the given conversation id. The response sets the following headers (to trigger file download).

- Content-Disposition: text/plain;charset=UTF-8
- Content-Type: attachment; filename="conversation.txt”

Glossary

List of terms and definitions.

- **Action flow**. A pipeline of requests and responses in the API. These flows are configurable in boost.ai's Admin Panel.
- **Admin Panel**. A web application to make action flows and parameters for VAs on boost.ai's platform.
- **Blocked conversation**. A conversation is blocked before an end user's consent is accepted. You can choose to block conversations until consent is accepted in the Admin Panel.
- **Conversation**. A list of elements and messages linked together by a conversation_id.
- **Conversation router**. A server state that keeps track of where requests from your chat client are routed to. Your requests might be routed to a VA or a person. In many cases, the person will be using boost.ai's Human Chat panel.
- **Conversation state**. Information about a specific conversation. The state of a conversation might change over time. The VA keeps a state of the conversation, and your client should keep at least a subset of the state to function properly.
- **Endpoint**. An endpoint in this API is a URL you can interact with by posting JSON messages.
- **Human Chat**. When a conversation is transferred to a person instead of the VA, the conversation is in Human Chat state.
- **Request**. A JSON object sent to the API, possibly expecting a response.
- **Response**. A JSON object with a list of data elements from the API.
- **User token**. A user token is a string that identifies an authenticated user. If provided, the VA will check if the token is valid through a dedicated endpoint. To set up an authenticated user flow, please contact boost.ai.

Author

Pål Thingbø - paal@boost.ai