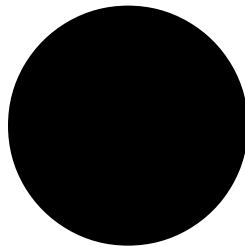




# LivePerson APIs

## Learn About LivePerson APIs

---



### *Consumer Interactions*

Learn about messsaging APIs for the consumer side.

[Learn More \(page 0\)](#)

# Protocol Overview

## API Endpoints

The API consists of two types of calls:

- Peripheral APIs - Domains Discovery, Token creation, Agent Profile, etc. In most cases these calls are done using REST.
- Messaging API - In this API, the WebSocket connection should be established.

## Messaging API WebSocket Connection

### Establishment

Refer to the [Getting Started \(page 7\)](#) section to see an example of how to establish this connection.

### Keep Alive

The server will close any idle connection. In order to keep the connection open, the client should send a message every 60 seconds. For this purpose, the client can use one of the following:

- WebSocket Ping/Pong Mechanism
- For clients that cannot send Ping messages (such as browsers): Utilize the applicative `.GetClock` request:  

```
{"kind": "req", "id": "1", "body": {}, "type": ".GetClock"}
```

### Error Handling

Upon disconnection, the client will receive a standard status code as defined by [rfc6455 \(https://tools.ietf.org/html/rfc6455#section-7.4\)](https://tools.ietf.org/html/rfc6455#section-7.4). Two custom status codes are used by the API:

- **4401** - A fresh token is required from the customer. When an authenticated connection is used, this code states that the request does not contain the required authorization token, or contains an expired token. This response asks the client to supply a new token from the customer.

- **4407** - For unauthenticated identities, this status code states that the client should ask the LivePerson IDP service to extend the validity of the current identity.

For any other status code, the client should wait a short period of time and then try to reconnect.

## Message Format

Every message sent by the client or by the server should be serialized to JSON format.

### Requests, Responses and Notifications

Each API message can be one of the following kinds:

- **Request** - A message sent from the client. The server will reply with a response message.
- **Response** - A message sent by the server in response to a client request message.
- **Notification** - A message sent from the server to the client, triggered by a server decision. The trigger can be based on a prior `Subscription` request made by the client, or on an implicit subscription made by the server.

The kind of message is denoted by the `kind` property in the top level of the JSON object, and can be any of the following: `req`, `resp`, `notification`. Below is an example of a request message:

```
{
  "kind" : "req",
  "type" : ".ams.cm.AgentRequestConversation",
  "id" : "hsjshka8162s",
  "body" : { "ttrDefName" : "NORMAL" }
}
```

### Message Types

Every message has a type. The structure of the body of the message can be changed for different types. The type of message is denoted by the `type` property in the top level of the JSON object. Below is an example of a message with the type of `.ams.cm.MyRequest`:

```
{
  "kind" : "req",
  "type" : ".ams.cm.MyRequest",
  "id" : "hsjshka8162s",
  "body" : { "ttrDefName" : "NORMAL" }
}
```

## Responses

Responses should be sent with some kind of correlation to the request that they are answering. For this purpose, every request message must state the `id` string for the request. The server will reply with the same `id` in the response message as the value of the `reqId` field in the top level JSON object of the response message. Below is an example of a response message to the request from the above section:

```
{
  "kind" : "resp",
  "reqId" : "hsjshka8162s",
  "type" : ".ams.cm.MyRequest$Response",
  "code" : 200,
  "body" : { "conversationId" : "hdjsdhksh2" }
}
```

The response message will also contain a `code` field. This field will be populated with the status code of the request. The values of this field will be taken from the [HTTP Semantics \(https://tools.ietf.org/html/rfc7231\)](https://tools.ietf.org/html/rfc7231).

## Subscriptions

In this pattern the client has to send some kind of request message to the server. In response, it will receive a response message with a success indication and a subscription ID. Following this, the server will start sending notifications with the subscription ID. ## API Client Requirements

## Client Future Compatibility

In order for the client to be compatible with future API changes, it should:

- Ignore any field that is not documented in the [API reference \(page 17\)](#). This includes existing fields that are not documented (for deprecation reasons), as well as new fields that will be added to the messages in newer versions.

- Ignore any enum fields that contain undocumented values.

### Single Element Arrays

An array containing a single element may be sent using the element itself instead of the array.

## Typical Frontend API Session

In this section we will provide an example API session.

### Initiation

The consumer initiates the messaging application on their device. At this point, the client should do the following:

1. Use the Authentication API to pass the external token and get an API token (JWT).
2. Open a WebSocket connection to the Interaction API.
3. Subscribe itself to the conversations metadata (in order to get the conversations list). This list may contain previously closed conversations along with existing active conversations. The list may be presented to the consumer.

The user may now want to [continue an existing conversation \(page 5\)](#) or [start a new conversation \(page 6\)](#).

### Continue an existing conversation

In the initiation process, the client got the conversation list, including the existing conversation (with OPEN state). To enable the consumer to continue this conversation, the client should do the following:

1. Subscribe to the conversation messaging events. If the client already has some of the messages of this conversation in its memory, it may want to subscribe only to the last messages sequence. This is to avoid sending over the network all the messages it already has.
2. Receive notifications about the events of this conversation from the server.
  - a. Text messages should be added to the presentation, ordered by their sequence.
  - b. Read/Accept events should add a graphical indication next to the text message they are referring to.

3. Keep receiving notifications from the server about new events currently being published by the other side.
4. Publish an `accept` event when a new text message is received to let other agents know that the message has been successfully delivered to the consumer device. When the message is presented to the consumer, the client should publish a `read receipt`.
5. Publish text messages when the consumer wants to send their own text.
6. Publish a `chat state` event when the consumer is typing, watching, or stops watching the conversation.

### Start a new conversation

In order to create a new conversation, the client should do the following:

1. Send a request to create a new conversation.
2. Get a notification about the new conversation.

The client should continue as described above for an [existing conversation \(page 5\)](#).

### Finish a conversation

The conversation can be closed either by the agent or by the consumer.

If the consumer wants to close the conversation by themselves, the client must send a request to update the conversation metadata and set the `state` to `close`. This can also be done by the agent.

The client will get a notification about this conversation with a `state` property set to `close`.

The client can present a CSAT survey to the consumer and send an `update conversation metadata` request with the CSAT information.

### Handle disconnection

When the API token expires, the WebSocket connection will be closed with `4401/4407` `closeReason`. The client should return to the [initiation \(page 5\)](#) step, issue a fresh token, and reestablish the connection.

For any other `closeReason`, the client should wait a few seconds and then try reconnecting.

# Getting Started

**Note:** This document is subject to change without notice.

In this tutorial you will create a new conversation with your contact center, publish text messages, and receive agent responses.

## Prerequisites

- [Docker](https://docs.docker.com/engine/installation) (<https://docs.docker.com/engine/installation>)
- LivePerson account enabled with two features: `Async_Messaging` and `Authenticated_Chat`. If you are not sure that your account is enabled with these two features, please contact LivePerson Support.

## Step 1 - Launch your Shell

We will use a docker image called `lp-shell` to run the shell for this tutorial. It is a Linux image equipped with `curl`, `wscat` (<https://www.npmjs.com/package/wscat2>), `jq` (<https://stedolan.github.io/jq/>) and several scripts.

After replacing the `__YOUR_ACCOUNT_ID__` with your account ID, type the following into your terminal:

```
docker run --env LP_ACCOUNT=__YOUR_ACCOUNT_ID__ -it eitanya/lp-shell
```

You should receive a shell line.

## Step 2 - Create a Token

In this demonstration we will create a new unauthenticated consumer identity. Run the following command:

```
LP_JWT=`curl -X POST https://$LP_IDP/api/account/$LP_ACCOUNT/signup | jq -r .jwt`
```

This will request the creation of a new identity. Extract the `JWT` from the response, and set it as a variable called `LP_JWT` in your shell. This new identity will be used in the next step.

To check the value inside the `LP_JWT`, type:

```
> echo $LP_JWT
eyJraWQiOiIwMDAwMSIsInR5cCI6IkpXVCIsImFsZyI6ImlJTMjU2In0.eyJzdWIiOiI1OTI2MzFhMC01ZmFlLTQ2YTdtODk2NC0xYzRjM2U3MjBhNGEiLCJhdWQiOiJhY2M6cWE1NzIyMTY3NiIsImZcyI6Imh0dHBzOlwvXC9pZHAubGl2ZXBlcnNvbWV0dGkiOiJF1QuMy08BVRH4ybgNfZxiTTQbAIWHDa4e24TohZGdqyZSv0Vlc4zgVP9wf0Svxbye_yyTx-Q_f8BB7Vy1ZdUYy0t_NK57bAxFTV8x4lx9fxAj_PJ0VvJncJjhn7JFu3s46WeKScFv28D78wJMgoHOKAKD-CacGzGtvFU9NuUQ
```

## Step 3 - Connect to the Messaging Service

Using the `JWT` we have created, we can open the connection to the UMS:

```
wscat -k 60 -H "Authorization:$LP_JWT" -c "wss://$LP_ASYNCMESSAGINGENT/ws_api/account/$LP_ACCOUNT/messaging/consumer?v=3"
```

In the following steps we will send JSON requests and receive responses and notifications through this connection. If the connection is closed, you can reconnect using the same `wscat` command.

## Step 4 - Create a New Conversation

Before creating the conversation, log into LiveEngage as an agent. In order for the agent to receive the messages, ensure that no other agent is logged into your account.

To request a new conversation, replace the `$LP_ACCOUNT` with your account ID in the following JSON object, and paste it into the connection from the previous step.

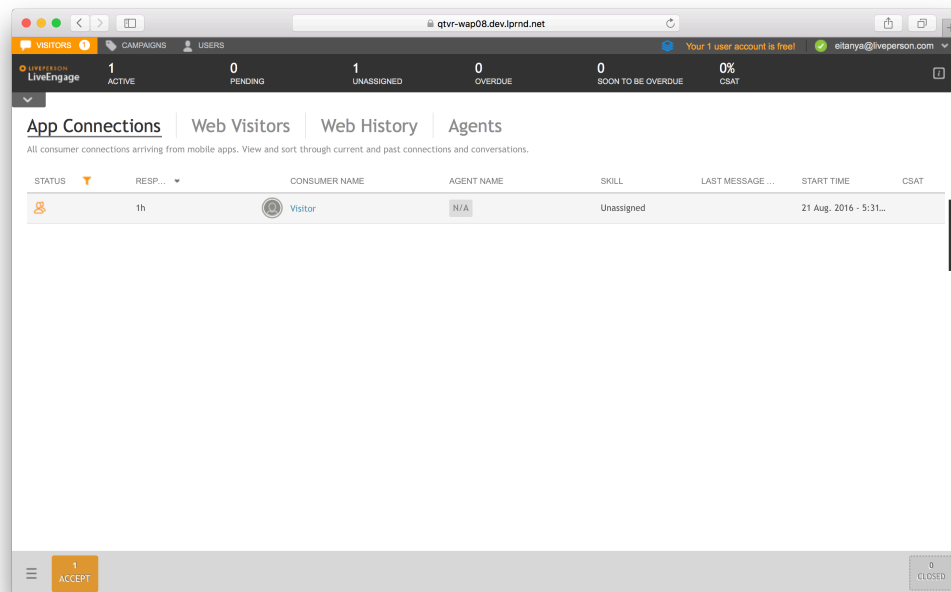
```
{"kind":"req","id":1,"type":".ams.cm.ConsumerRequestConversation","body":{}}
```

In response, you will receive the ID of the new conversation.



```
{
  "kind": "resp",
  "reqId": "1",
  "code": 200,
  "body": {
    "conversationId": "7507be78-60ef-4468-b3b1-baa47fbbeea21"
  },
  "type": ".ams.cm.types.RequestConversation$Response"
}
```

Open the Agent Workspace (from Step 1), and accept the incoming request by clicking the **Accept** button.



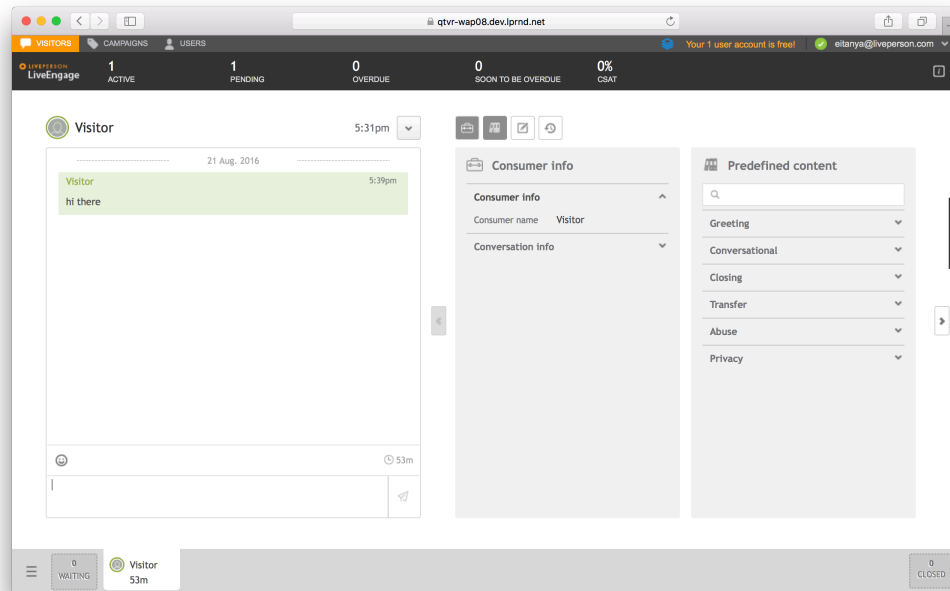
## Step 5 - Publish a Text Message

In order to publish content to a conversation, substitute the `__YOUR_CONVERSATION_ID__` with the `consersationId` you got in the response of Step 4, and paste it into the opened WebSocket.

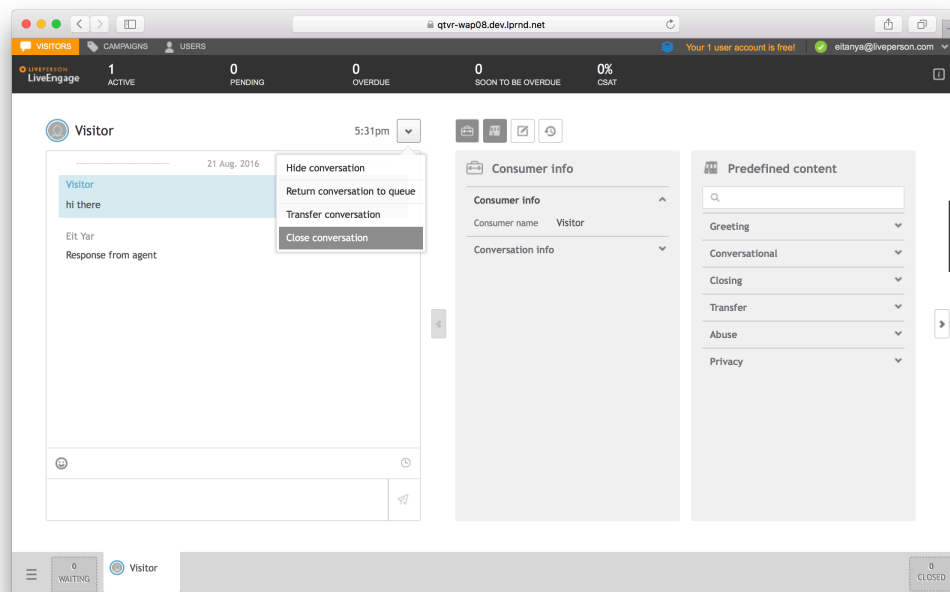
```
{"kind": "req", "id": 2, "type": ".ams.ms.PublishEvent", "body": {"dialogId": "__YOUR_CONVERSATION_ID__", "event": {"type": "ContentEvent", "contentType": "text/plain", "message": "hi there"}}
```

 (page 0)

The published message will be displayed on the agent side:



Now close the conversation from the Agent Workspace using the  
Close conversation menu item:



## Step 6 - Record LP\_JWT to Next Tutorials


In order to use the consumer token (LP\_JWT) in the following tutorials, we will put a file on your machine. Create a new file outside of the `lp-shell` and paste the following content into it:

```
LP_ACCOUNT=qa57221676                                     #
put here your account id

# Put here the jwt you have created in this tutorial
# You can use echo $LP_JWT from your lp-shell to view it
LP_JWT=eyJraWQiOiIwMDAwMSIsInR5cCI6IkpXVCIsImFsZyI6IjEwInQ.eyJzdWIiOiJmMDY4ZTllZi1lMzkzLTQxYTEtYmMyYy1hOTZhOWZmMGY2ZmIiLCJhdWQiOiJhY2M6cWE1NzIyMTY3NiIsImIzcyI6Imh0dHBzOlwvXC9pZHAubGl2ZXBlcnNvbi5uZXQiLCJleHAiOjE4ODIwMjMzISImhdCI6MTQ3MTc4MzU5MnQ.
HUJe1CZzqzRoJJxoTlL_vDvRa1KIEJJRt2MEhY-aFq__V6lrN-ebrRxydoz
m-gjbpMecKiZDZZiJPw3hf560iKbW-gK1AzsfHxiPrxMdg_TRZqsNhXui_7k579
IpfAvKSdgQHZ5uLfGq2XtQNfBdvKWPCIAfW8mJ7oZT-aNMhjE
```

## What's next

You can now close the `lp-shell` and move on to the [next tutorial \(page 12\)](#).

Explore API messages in the [API Reference \(page 17\)](#) page. In order to build your own messages, you can use the `message builder`  [\(page 0\)](#)

## Get Messages

In this tutorial we will demonstrate the mechanism that enables the consumer to get messages sent by the agent. These messages may be sent while the consumer is connected or offline.

### Prerequisites

At the end of the [Getting Started \(page 7\)](#) tutorial, we created the `lp.env` file. In order to launch `lp-shell` with your previous settings, type the following:

```
docker run --env-file lp.env -it eitanya/lp-shell
```

You should get a shell line.

Make sure that you have closed any previous conversation sessions by clicking `Close conversation` in the Agent Workspace.

### Step 1 - Find Your Consumer ID

When you get messages from the server, you must identify which messages were published by you, and which were published by the agent. To do so, you must know your `consumerId`. To find out your `consumerId`, refer to the content of the `LP_JWT` by base64-decoding the middle part of the JWT (between the two periods):

```
> echo $LP_JWT | sed -e 's/.*\.(.*)\..*/\1/' | base64 --decode
```

You will now be able to see the content of the JWT:

```
{
  "sub": "19f98cd2-63b9-42e4-bc0d-9222b86574e1",
  "aud": "acc:qa57221676",
  "iss": "https://idp.liveperson.net",
  "exp": 1879957364,
  "iat": 1469717324
}
```

The `sub` property contains your `consumerId`.

**Note:** If you encounter issues when using the shell `base64` command, you can use online tools such as [base64decode.org](https://www.base64decode.org/) (<https://www.base64decode.org/>).

## Step 2 - Create a Conversation

Open the WebSocket connection using the following:

```
wscat -k 60 -H "Authorization:$LP_JWT" -c "wss://$LP_ASYNCMESSAGING/ws_api/account/$LP_ACCOUNT/messaging/consumer?v=3"
```

Replace the `__LP_ACCOUNT__` with your account ID, and then send a request on the connection to create a new conversation with the following JSON object:

```
{"kind": "req", "id": 1, "type": ".ams.cm.ConsumerRequestConversation", "body": {}}
```

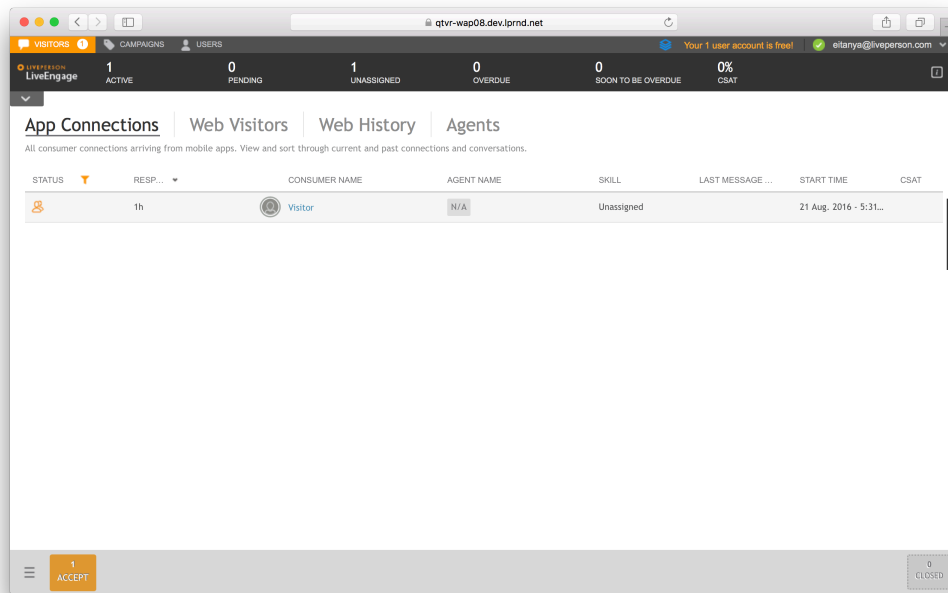
In response, you will get the ID of the new conversation.

```
{
  "kind": "resp",
  "reqId": "1",
  "code": 200,
  "body": {
    "conversationId": "7507be78-60ef-4468-b3b1-baa47fbcea21"
  },
  "type": ".ams.cm.types.RequestConversation$Response"
}
```

Write down the `conversationId` from the response. We will need it in the next steps.

## Step 3 - Agent sends Messages

In this stage, switch to the Agent Workspace from Step 1, and accept the ring of the incoming request. Click the blinking `Accept` button.



Type a few messages in the Agent Workspace.

## Step 4 - Subscribe to Conversation Content

In order to get existing or new messages from the agent side, the consumer should subscribe to the content of the conversation. Substitute the `__YOUR_CONVERSATION_ID__` with the `consersationId` you got in the response in the previous step, and paste it into the opened WebSocket.

```
{
  "kind": "req",
  "id": "22",
  "body": {
    "fromSeq": 0,
    "dialogId": "__YOUR_CONVERSATION_ID__",
    "type": ".ams.ms.SubscribeMessagingEvents"
  }
}
```

In response, you will get a subscription success message:

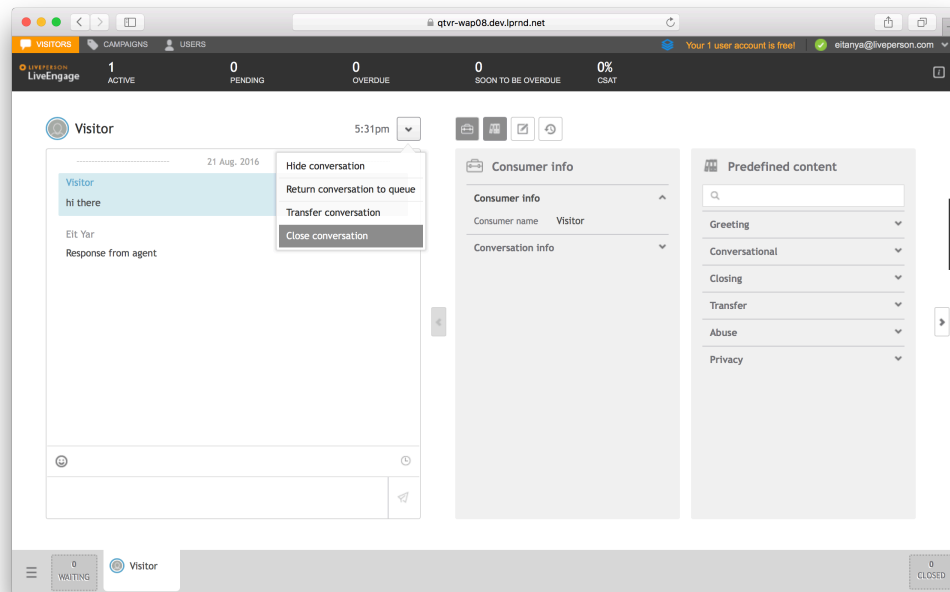
```
{
  "kind": "resp",
  "reqId": "22",
  "code": 200,
  "body": {},
  "type": ".ams.GenericSubscribe$Response"
}
```

You will now get all the existing content of the conversation:

```
{
  "kind": "notification",
  "body": {
    "changes": [
      {
        "sequence": 0,
        "originatorPid": "734d9867-40e3-52b9-a401-07e877676d64",
        "serverTimestamp": 1477840831162,
        "event": {
          "type": "ContentEvent",
          "message": "message from the agent",
          "contentType": "text/plain"
        },
        "dialogId": "__YOUR_CONVERSATION_ID__"
      }
    ],
    "type": ".ams.ms.MessagingEventNotification"
  }
}
```

You will notice that you get all the messages that were published by you, or by the agent. In order to find which messages were published by you, refer to the `originatorPid` field. Messages that were published by you will have a value equal to the `consumerId` you found in Step 1, while messages from the agent will have a different `originatorPid`.

Now close the conversation from the Agent Workspace by clicking **Close conversation**.





# API Reference

## Connection Establishment



From lp-shell:














```
wscat -k 60 -H "Authorization:$LP_JWT" -c "wss://$LP_ASYNCMESSAGINGENT/ws_api/account/$LP_ACCOUNT/messaging/consumer?v=3"
```






Client properties can be added to the connection URL as query params. The full list of supported client properties can be found [here \(page 0\)](#).

Where LP\_JWT is your token.

## Interaction Commands

You can use the `message builder`  to build and explore the structure of the API messages. The following table summarizes the API messages and provides some examples .

Category	Name	Request	Response	Notifications
Message Builder		 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>
Conversations	RequestConversation	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	<i>n/a</i>
	<b>UpdateConversationField</b> Close, update CSAT.	 <a href="#">(page 0)</a>  <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	<i>n/a</i>
Conversation Metadata	SubscribeExConversations	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>
	UnsubscribeExConversations	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	<i>n/a</i>

Category	Name	Request	Response	Notifications
Messages	<b>PublishEvent</b> Send text, read/accept and presence events.	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	<i>n/a</i>
	SubscribeMessagingEvents	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>	 <a href="#">(page 0)</a>

## Agent Public Profile

```
curl https://$LP_ACCDNDOMAIN/api/account/$LP_ACCOUNT/configuration/le-users/users/$LP_AGENT_PID
```

## Other APIs

- [Authentication \(page 0\)](#)