

Management and Data Reporting for Electronic Temperature Sensors



ZEBRA

Developer Guide

2023/09/10

ZEBRA and the stylized Zebra head are trademarks of Zebra Technologies Corporation, registered in many jurisdictions worldwide. All other trademarks are the property of their respective owners. ©2023 Zebra Technologies Corporation and/or its affiliates. All rights reserved.

Information in this document is subject to change without notice. The software described in this document is furnished under a license agreement or nondisclosure agreement. The software may be used or copied only in accordance with the terms of those agreements.

For further information regarding legal and proprietary statements, please go to:

SOFTWARE: zebra.com/linkoslegal

COPYRIGHTS: zebra.com/copyright

PATENTS: jp.zebra.com

WARRANTY: zebra.com/warranty

END USER LICENSE AGREEMENT: zebra.com/eula

Terms of Use

Proprietary Statement

This manual contains proprietary information of Zebra Technologies Corporation and its subsidiaries ("Zebra Technologies"). It is intended solely for the information and use of parties operating and maintaining the equipment described herein. Such proprietary information may not be used, reproduced, or disclosed to any other parties for any other purpose without the express, written permission of Zebra Technologies.

Product Improvements

Continuous improvement of products is a policy of Zebra Technologies. All specifications and designs are subject to change without notice.

Liability Disclaimer

Zebra Technologies takes steps to ensure that its published Engineering specifications and manuals are correct; however, errors do occur. Zebra Technologies reserves the right to correct any such errors and disclaims liability resulting therefrom.

Limitation of Liability

In no event shall Zebra Technologies or anyone else involved in the creation, production, or delivery of the accompanying product (including hardware and software) be liable for any damages whatsoever (including, without limitation, consequential damages including loss of business profits, business interruption, or loss of business information) arising out of the use of, the results of use of, or inability to use such product, even if Zebra Technologies has been advised of the possibility of such damages. Some jurisdictions do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you.

Contents

Overview.....	5
Workflow.....	5
Getting Started.....	7
API Authentication.....	7
Using a Simple Key.....	7
OAuth Client Credentials.....	9
OAuth Authorization Code.....	10
Enrolling Devices.....	11
Enrolling a ZS300 Sensor.....	11
Listing the Sensors.....	13
Listing the ZS300 Sensors.....	13
Creating a Task.....	15
Associating a Sensor with a Task.....	16
Associating a Sensor with a Task.....	16
Performing a Simple Task.....	17
Stopping a Task.....	17

- Using Webhook Subscriptions.....19**
 - Understanding Webhook Concepts..... 19
 - Understanding Webhooks and APIs..... 19
 - Knowing When to Use Webhooks..... 21
 - Creating a Webhook Subscription.....22
 - Starting a Webhook Subscription..... 23
 - Stopping a Webhook Subscription..... 24
 - Understanding Webhook Outputs..... 25

- Displaying Full Event Record for Task ID..... 27**
 - Using the Developer Portal Interactive Documentation..... 27
 - Sorting, Filtering, and Pagination..... 29
 - Using an API Test Tool..... 29

- Downloading AIDL Files.....31**

Overview

This guide is for developers of applications that integrate data from the Zebra ZS300 Sensor.

The Management for Electronic Temperature Sensor Application Programming Interfaces (APIs) provide the ability to configure sensors for use. This includes enrolling sensors, creating tasks, configuring sensors, adding sensors to tasks, associating asset IDs with tasks, and stopping tasks.

The Data Reporting for Electronic Temperature Sensor APIs are designed to provide data for task reports. Every sensor reading and monitored event that occurs during the use of the sensor assigned tasks are recorded and stored for seven years.

These APIs allow you to view that history making them useful for reconciliation and final disposition determinations. This set of APIs are designed to provide data that can be used to create post-task reports. The log request is a raw request to the long-term data storage system.

Every temperature reading captured by the sensor is listed as a separate event. This means that every sensor can have many thousands of events associated with it. The system also captures other major events in the use of the sensor. The APIs provide means of filtering and paginating this data.

Workflow

This table describes steps needed to achieve a simple use case for using Zebra Temperature Sensors.

Step	API	Comments
1. Enroll the sensor	POST /devices/sensor-enrollments	Use the serial number on the sensor.
2. List sensors associated with a tenant	GET /devices/environmental-sensors	Verify that the sensor has been enrolled. Optional filters are available.
3. Create a task	POST /environmental/tasks	Configure the temperature recording thresholds and start method. Repeat for each task.
4. Create a Webhook subscription		User receives only one Webhook notification per sensor per task.
5. Associate the sensor with the task	POST /environmental/tasks/{taskId}/sensors	Use the Task ID returned when the task was created and the Sensor ID returned when verifying sensor enrollment. Repeat for each sensor or associate multiple sensors at the same time. The task starts when the sensor receives the task.

Overview

Step	API	Comments
6. Add an asset to a task	POST /environmental/tasks/{taskId}/assets	Use the task ID returned when the task was created
7. Stop the task	POST /environmental/tasks/{taskId}/stop	Stop recording temperatures.
8. Retrieve details for the task	GET /environmental/tasks/{taskId}	Use the Task ID returned when the task was created.
9. Retrieve all tasks	GET /environmental/tasks	Optional filters are available.
10. Retrieve alarms for the task	GET /environmental/tasks/{taskId}/alarms	Use the Task ID returned when the task was created. Optional filters are available.
11. Retrieve sensor read-event logs for the task	GET /data/environmental/tasks/{taskId}/log	Use the Task ID returned when the task was created. Data may not be immediately available.

Getting Started

This section details the available authentication methods needed to gain access to all Zebra APIs.




NOTE: The Client Key is sometimes referred to by several names: App Key; Consumer Key; or Client Key. For the purposes of this guide, we will use Client Key.

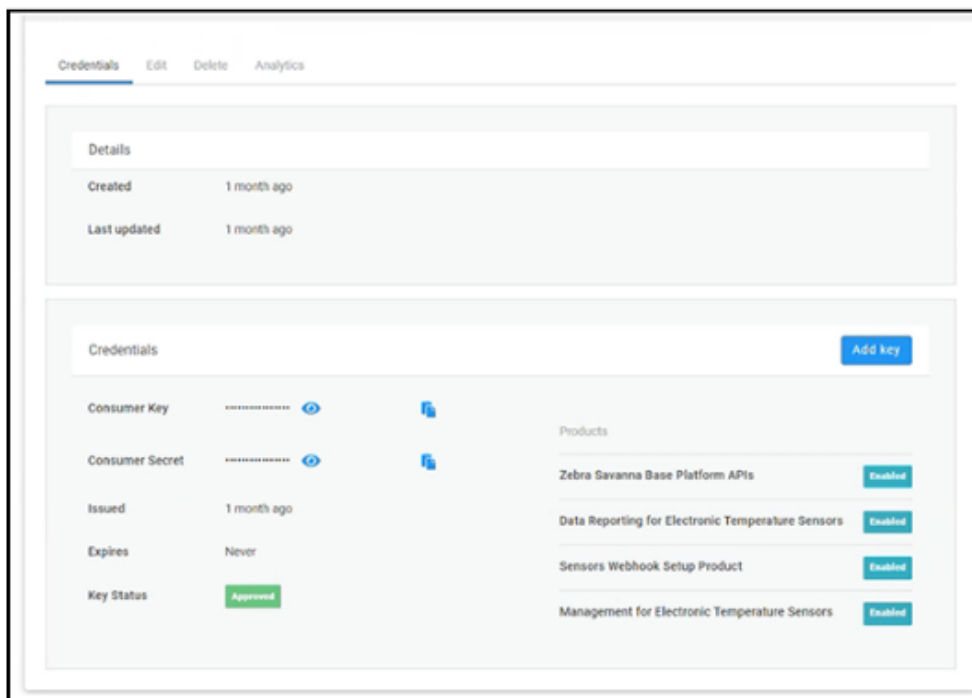
API Authentication

Access to all Zebra APIs needs to be authenticated. There are several methods available for Authentication with Zebra APIs. Once you have been granted access to the Electronic Temperature Sensor APIs, you will use your client key on your App page (developer.zebra.com/user/apps) of the Zebra Developer Portal. It is very important that you keep this key safe and secure from your code, repositories, or other individuals. Do not give or share your full key with anyone.

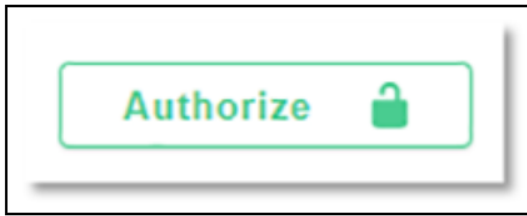
Using a Simple Key

Most of the APIs can be authenticated by providing the client key in the headers under the `apikey` parameter. This is only recommended for initial testing and Proof of Concept work. This is the most insecure method and should never be used outside of the developer's control.

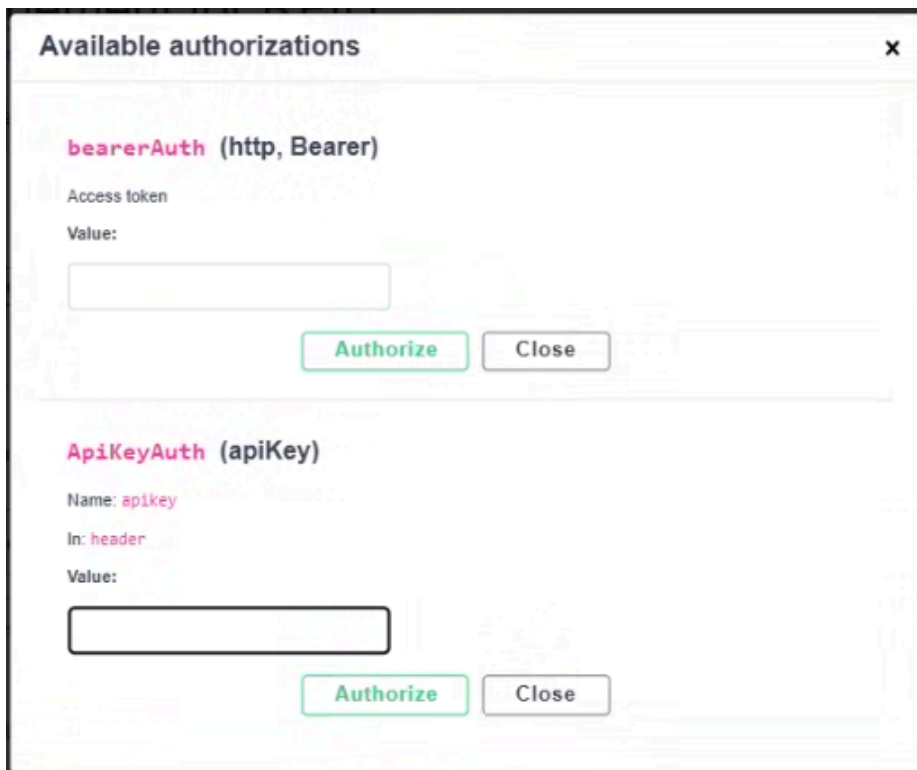
1. Click  to copy the client key.



2. Go to the desired API page.
3. Click **Authorize**.



4. Paste the Client Key into the **ApiKeyAuth** text box and click **Authorize**.

A screenshot of a dialog box titled "Available authorizations" with a close button (X) in the top right corner. The dialog contains two sections. The first section is for "bearerAuth (http, Bearer)", showing "Access token" and "Value:" with an empty text input field, and "Authorize" and "Close" buttons. The second section is for "ApiKeyAuth (apiKey)", showing "Name: apikey", "In: header", and "Value:" with an empty text input field, and "Authorize" and "Close" buttons.

OAuth Client Credentials

An OAuth bearer token can be created by providing the client key and client secret. You will find a client key and secret on the App Page of the Zebra Developer Portal. All Zebra Bearer tokens last for one hour but can be refreshed using the previous token. Client Credentials grant type should only be used from a client app's server, never directly from a client app as this may provide open access to client keys.

Generate token: developer.zebra.com/apis/oauth-client-credentials

To verify, call the API above and get a token. If the token returns with abc123, then enter that in the **Value** field in the **Available Authorizations** dialog and click **Authorize**.

The image shows a dialog box titled "Available authorizations" with a close button (x) in the top right corner. It contains two sections for different authorization methods:

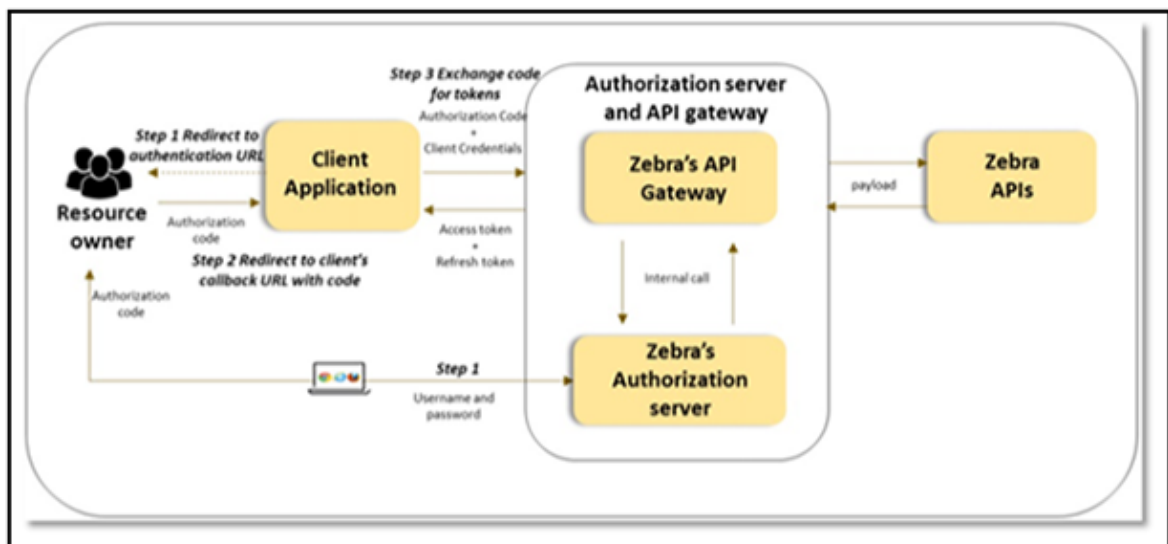
- bearerAuth (http, Bearer)**
 - Access token
 - Value:
 - Input field containing "Bearer abc123"
 - Buttons: "Authorize" (green) and "Close" (grey)
- ApiKeyAuth (apiKey)**
 - Name: apiKey
 - In: header
 - Value:
 - Input field (empty)
 - Buttons: "Authorize" (green) and "Close" (grey)

OAuth Authorization Code

The primary method for creating a customer-specific Bearer token is the Authorization Code grant type. This is the standard 3-legged OAuth used in many web applications. It is considered one of the most secure methods of getting a token. All Zebra Bearer tokens last for one hour but can be refreshed using the previous token.

Authorization code grant type is a multi-step process. First you redirect to the IDP login and provide a redirect URI. After the customer logs in, the IDP will send the authorization code to the client app URI. Authorization codes last for 10 seconds. The client app calls the token API with the code to get a bearer token.

Figure 1 Authorization Code Workflow



Go to developer.zebra.com/apis/oauth-authorization-code-0 for more information on OAuth Authorization Code.

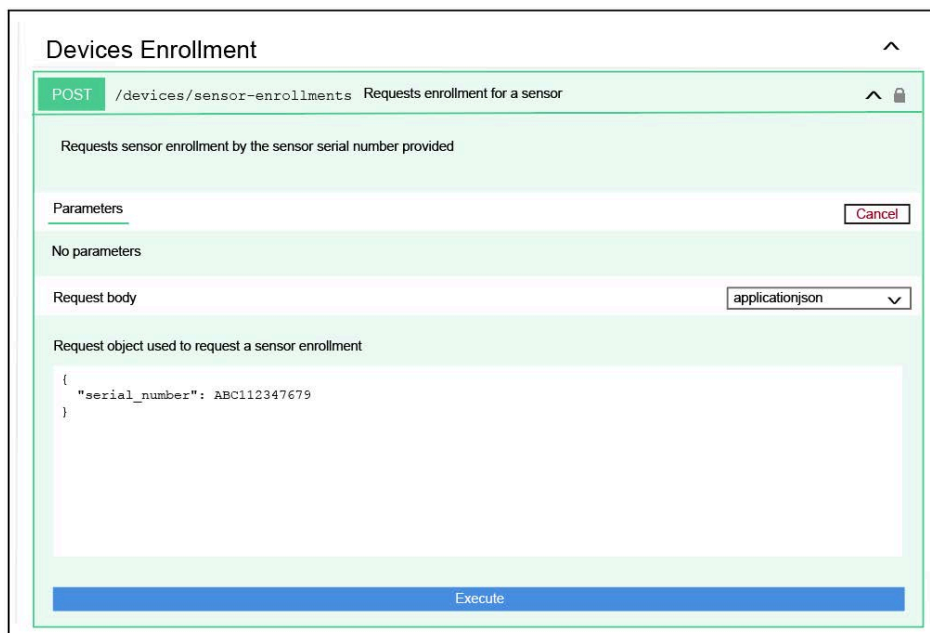
Enrolling Devices

This section describes the steps to enroll a ZS300 Sensor via the Zebra Developer Portal.

Enrolling a ZS300 Sensor

To enroll a ZS300 Sensor in the Zebra Developer Portal:

- ZB200 Bridge or an Android v8.1 or later mobile device running the Android Sensor Discovery Service
 - Access to an ethernet port and ethernet cable if using a bridge, and there is no access to a wireless network or user is using ZSFinder.
 - One or more ZS300 Sensors.
 - A Client Key
1. Navigate to the Zebra Developer Portal at developer.zebra.com.
 2. Login to the Portal.
 3. Navigate to the [Management for Electronic Temperature Sensors API](#) page.
 4. Click **Authorize** and enter your Client Key.
 5. Click **Close**.
 6. Expand the POST/devices/sensor-enrollments method screen.
 7. Click **Try It Out**.
 8. Enter the serial number located on the front of the ZS300 Sensor in the JSON body.



The screenshot displays the 'Devices Enrollment' API endpoint configuration in the Zebra Developer Portal. The endpoint is a POST request to `/devices/sensor-enrollments` with the description 'Requests enrollment for a sensor'. The request body is set to 'application/json' and contains a JSON object with a single key-value pair: `{ "serial_number": "ABC112347679" }`. The interface includes a 'Parameters' section with a 'Cancel' button, a 'Request body' section with a dropdown menu set to 'application/json', and an 'Execute' button at the bottom.

```
POST /devices/sensor-enrollments Requests enrollment for a sensor

Requests sensor enrollment by the sensor serial number provided

Parameters [Cancel]

No parameters

Request body [application/json]

Request object used to request a sensor enrollment

{
  "serial_number": "ABC112347679"
}

Execute
```

- Click **Execute**. If successful, an HTTP Status Code 200 displays, the response body is an empty JSON body, and the device is scheduled to be enrolled.

Code	Description	Links
200	A successful response. Media type <input type="text" value="application/json"/> Controls Accept header . Example Value Schema <div style="background-color: black; color: white; padding: 5px; text-align: center;">0</div>	No links
400	An unexpected error response	No links

- Hold down the button on the front of the Sensor until the amber LED blinks. a The device enrolls after one or two minutes.



- Proceed with Listing the Sensors.

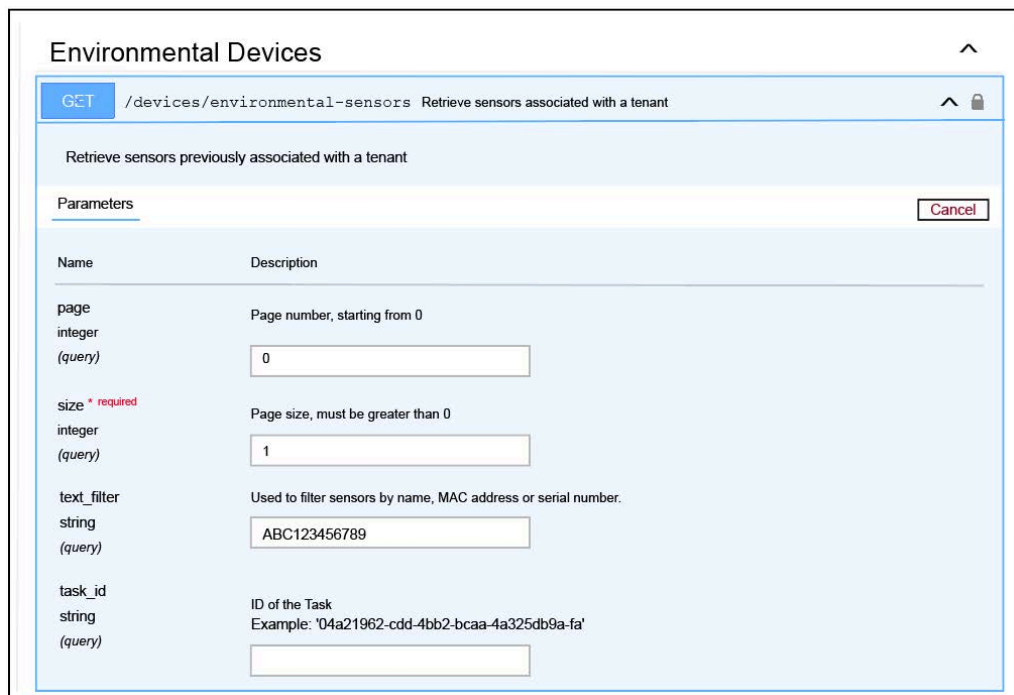
Listing the Sensors

This section describes how to list the ZS300 Sensors via the Zebra Developer Portal.

Listing the ZS300 Sensors

Before performing the following steps, at least one ZS300 Sensor must be enrolled.

1. Navigate to the [Management for Electronic Temperature Sensors API](#) page.
2. Click **Authorize** and enter your Client Key.
3. Click **Close**.
4. Expand the GET/devices/environmental-sensors method screen.
5. Click **Try It Out**.
6. In the page number field, enter 0.
7. Enter 1 in the page size field. If more than one sensor is enrolled, enter the appropriate number.
8. Enter the serial number of the Sensor in the text filter field.
9. Delete the default values in the **task_id**, **enrolled_after** and **enrolled_before** fields if those are not needed.



The screenshot shows the 'Environmental Devices' API interface. The endpoint is GET /devices/environmental-sensors, which retrieves sensors associated with a tenant. The interface includes a 'Parameters' section with the following fields:

Name	Description
page integer (query)	Page number, starting from 0 <input type="text" value="0"/>
size * required integer (query)	Page size, must be greater than 0 <input type="text" value="1"/>
text_filter string (query)	Used to filter sensors by name, MAC address or serial number. <input type="text" value="ABC123456789"/>
task_id string (query)	ID of the Task Example: '04a21962-cdd-4bb2-bcaa-4a325db9a-fa' <input type="text"/>

Listing the Sensors

10. Click **Execute**. If successful, a response body with an HTTP Status Code of 200 is returned. A JSON body is included in the response which contains details of the currently enrolled Sensor.

```
{
  "sensors": [
    {
      "id": "6f6260ef-45d3d4a81-94b7-9fd19e820f6b",
      "mac_address": "00074DDDDA51",
      "serial_number": "DLJ222900904",
      "model": "ZS300",
      "manufacturer": "Zebra Technologies",
      "hardware_revision": "2",
      "firmware_revision": "D01.0.1.779",
      "battery_level": 97,
      "name": "ZS300_DLJ222900904",
      "status": "SENSOR_STATUS_STOPPED",
      "alarm_count": null,
      "first_seen": "2023-05-04T16:10:00.989720Z",
      "last_updated": "2023-05-05T23:05:23.537368Z",
      "notes": "",
      "certificate_url": "https://storage.googleapis.com/spg-zpc-d-envirovue-certs/certificates/ZS300_Sensor_Certificate_of_Conformance_RevB.pdf",
      "most_recent": {
        "task_id": "9d8f121e-61e1-4974-8863-cfcc9d4e95fc",
        "sensor_task_id": "f95ceb4f-79b0-43f7-ab4a-80757fd490cc",
        "sensor_task_status": "SENSOR_TASK_STATUS_COMPLETED",
        "alarm_count": 0
      },
      "requested": {
        "task_id": "",
        "sensor_task_id": "",
        "sensor_task_status": "SENSOR_TASK_STATUS_UNSPECIFIED",
        "alarm_count": null
      },
      "certificate_type": "CERTIFICATE_TYPE_CONFORMANCE",
      "unverified": {
        "last_date_time": "2023-05-05T23:05:23.567225Z",
        "last_temperature": 327.67,
        "last_alarm": false
      }
    }
  ]
}
```



NOTE: Occasionally, the 327.67 temperature reading may appear in the Last Temperature field. It is an invalid reading. When a sensor comes out from the deep sleep, it takes up to a minute before it starts sampling. During this time, the sensor doesn't have a valid temperature, so a value of 7FFF is used which produces the temperature reading of 327.67.

11. Save the ID of the Sensor for the next step.

Creating a Task

This section describes how to create a task via the Zebra Developer Portal

1. Navigate to the [Management for Electronic Temperature Sensors API](#) page.
2. Click **Authorize** and enter your Client Key.
3. Click **Close**.
4. Expand the POST/environmental/tasks method screen.
5. Click **Try It Out**.
6. Enter the sample JSON in the Request Object field.
7. Enter a unique name in the name field.

```
{
  "task_from_details": {
    "task_details": {
      "name": "Your tenant",
      "interval_seconds": 15,
      "loop_reads": true,
      "start_immediately": {},
      "sensor_type": "SENSOR_TYPE_TEMPERATURE",
      "alarm_low_temp": 15,
      "alarm_high_temp": 30,
      "low_duration_seconds": 30,
      "high_duration_seconds": 30
    }
  }
}
```

8. Click **Execute**. If successful, a response with an HTTP Status Code of 200 is returned. The task ID is returned in the response body.

```
{
  "id": "abe2c7d0-a8f8-4017-8958-abe2c7d0204f"
}
```

Associating a Sensor with a Task

This section describes how to associate a ZS300 Sensor with a task via the Zebra Developer Portal.

Associating a Sensor with a Task

The ID on an enrolled Sensor must be available before attempting the following procedure.



NOTE: The user can associate multiple sensors with a task at a time

1. Expand the POST/environmental/task/{taskId}/sensors method screen.
2. Click **Try It Out**.
3. Enter the returned task ID in the in the taskId path field.
4. Enter the returned sensor ID from Listing the Sensor in the Sensor ID array in the Request Object field.
5. Click **Execute**. If successful, a response with an HTTP Status code of 200 is returned. A JSON object with the sensor_id and sensor_task_id association is returned.

```
{
  "associated_sensors": [
    {
      "sensor_id": "83f87911-2184-4f26-91c5-83f87911c1b4",
      "sensor_task_id": "abe2c7d0-a8f8-4017-8958-abe2c7d0204f"
    }
  ]
}
```

6. Press the button on the Sensor and hold for 10 seconds until the LED blinks orange. After the Sensor blinks green, the task has started.



NOTE: If the Sensor LED does not blink green, move the Sensor closer to the Bridge.

Performing a Simple Task

This section describes how to perform a simple task via the Zebra Developer Portal.

Stopping a Task

At least one ZS300 Sensor must be enrolled, a task created, and one or more Sensors associated with a task before attempting the following procedure.

1. Repeat the [Listing the Sensors](#) procedure.
2. Note that the status field of the Sensor is SENSOR_STATUS_STOPPED.



NOTE: If you repeat the List the Sensors procedure again, the status changes to SENSOR_STATUS_ACTIVE.

3. Allow the task to run for at least five minutes.
4. Move the Sensor around to different temperature conditions.
5. After five minutes, stop the task by expanding the POST/environmental/tasks/{taskId}/stop window.
6. Click **Try It Out**.
7. Enter the taskId in the path field.

The screenshot shows a REST client window with the following details:

- Method: GET
- Path: /environmental/tasks/{taskId}/stop
- Operation Name: Stop a Task
- Description: Stop the task specified by the taskId provided
- Parameters section with a "Cancel" button.
- Table of parameters:

Name	Description
taskId * required	Required ID of the Electronic Temperature Sensor Task you wish to stop
string	Example: '04a21962-cdd-4bb2-bcaa-4a325db9a-fa'
(path)	

The input field for the path parameter contains the value: 04a21962-cdd-4bb2-bcaa-4a325db9a-fa

At the bottom of the window is a blue "Execute" button.

8. Click **Execute**. If successful, a response body with an HTTP Status Code of 200 is returned and the task is scheduled to stop. When all sensors associated with the task are within range of the bridge, then the task is stopped.

```
{
  "task": {
    "id": "abe2c7d0-a8f8-4017-8958-abe2c7d0204f",
    "sensor_count": "1",
    "status": "TASK_STATUS_STOP_PENDING",
    "sensor_task_status_overview": {
      "stop_pending": "1"
    },
  },
  "started": "2022-11-04T20:22:33.303659Z",
  "taskDetails": {
    "name": "Ryan Task 1",
    "created": "2022-11-04T20:10:06.083344Z",
    "updated": "2022-11-04T20:30:10.762285Z",
    "interval_minutes": 0,
    "interval_seconds": 15,
    "loop_reads": true,
    "start_immediately": {},
    "sensor_type": "SENSOR_TYPE_TEMPERATURE",
    "alarm_low_temp": 15,
    "alarm_high_temp": 30,
    "low_duration_seconds": 30,
    "high_duration_seconds": 30
  },
  "alarm_count": "0"
}
```

9. Repeat the [Listing the Sensors](#) procedure to verify that the status field has returned to the SENSOR_STATUS_STOPPED status and the sensor task status is SENSOR_TASK_STATUS_COMPLETED.

Using Webhook Subscriptions

Webhooks enable real-time notifications and data updates. Instead of one application making a request to another to receive a response, a webhook is a service that allows one program to send data to another as soon as a particular event occurs.

Understanding Webhook Concepts

To help you better understand the content in this guide, the following definitions explain Webhook concepts:

- **Webhook:** A single event message. Zebra sends a webhook to your application's webhook subscription endpoint. A webhook contains a JSON payload in the body and metadata in the headers.
- **Webhook Subscription:** This is a persisted data object within your application that defines the webhook subscription endpoint. Your webhook subscription should also have logic for handling data that will be posted to your endpoint from Zebra.
- **Webhook Subscription Endpoint:** The destination where Zebra sends webhooks for the specified event.
- **Webhook Listener:** An app that provides the endpoint for a webhook subscription to which to send event messages.

Understanding Webhooks and APIs

APIs enable two-way communication between software applications. Requests, also called polling, drive communication between the two applications. On the other hand, webhook subscriptions allow for one-way data sharing triggered by events rather than requests. In the following two images, you can see how webhooks work and how they can provide more efficient, cost-effective, and timely data access than API polling.

Figure 2 Polling with APIs

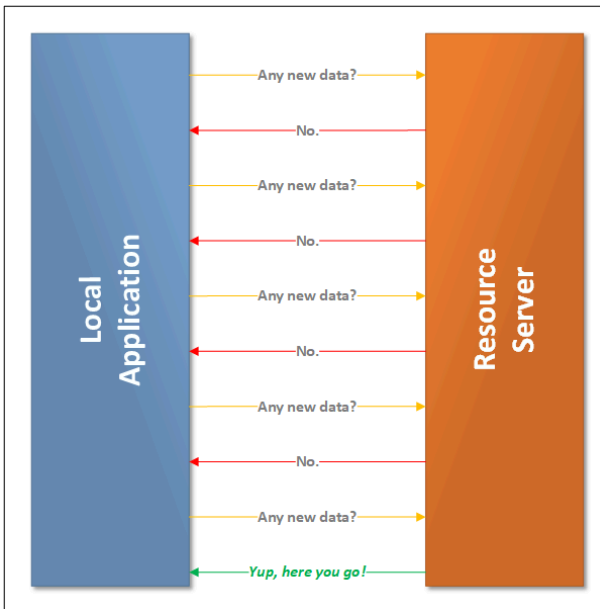
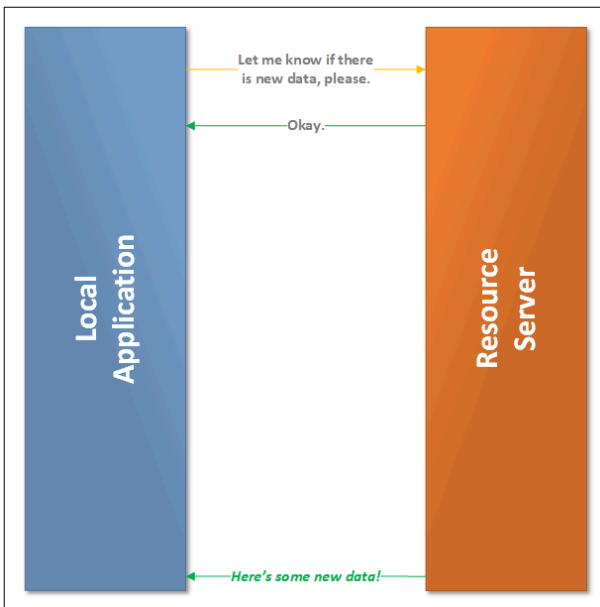
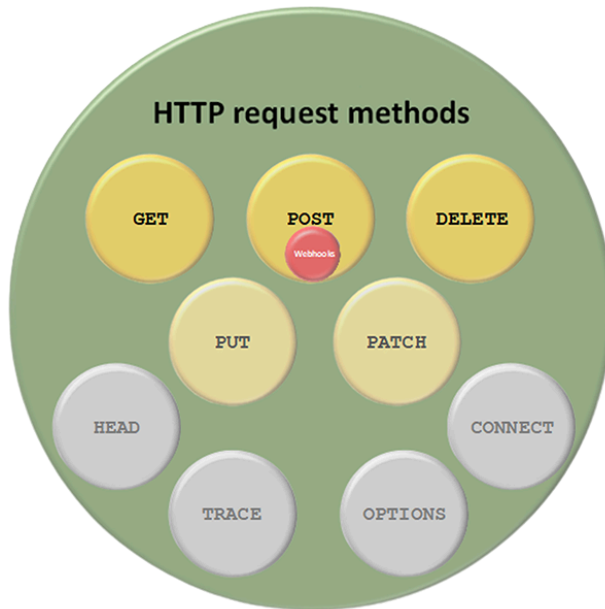


Figure 3 Webhooks



For the most part, webhooks can be seen as a subset of the POST method, as illustrated in the image below. This is because webhooks perform a POST call to an API endpoint you set up in your environment. When setting up your webhook subscription, designate the endpoint you created as the place the data will be POSTed when sent.

Figure 4 HTTP Request Methods



Knowing When to Use Webhooks

Webhooks are a simplified model of communication thus, you should use webhooks when you require the following:

- Real-time one-way communication (from source to destination)
- A non-persistent connection between the two systems' communication
- Immediate response to an event from a SaaS application that supports webhooks
- Use of the push model to immediately push updates
- One-to-one communication

Manage exceptions: You need to manage any exception that will occur, such as shipments falling outside of regulated temperature ranges.

Creating a Webhook Subscription

This section describes how to create a webhooks subscription via the Zebra Developer Portal for the purpose of capturing temperature excursion events.

You must provide a server with an HTTPS-capable static endpoint to receive data from Zebra. The endpoint must:



NOTE: It is recommended that webhooks are created after the task is created but before sensors are associated with the task, otherwise an alarm may occur before the webhook has been created. If that occurs, the webhook notification will not be received.

- Accept POST requests. Zebra sends customer data to the endpoint you designate in POST requests.
- Accept JSON data. Zebra sends data in JSON form. The application/json content-type will deliver the JSON payload directly as the body of the POST request.
- Use HTTPS. Zebra transmits potentially sensitive data on behalf of customers, and HTTPS is the first step in ensuring their data stays safe.

Prerequisites

- Create a webhook endpoint within your application(s)
 - Add logic to handle Zebra events. For [PLATFORM or SERVICE] subscriptions, these include: [TEMP DATA]
 - Test your webhook endpoint to confirm that it's working as expected
1. Ensure your application key for usage of this API is available. See the [Getting Started Guide](#) for more details.
 2. Create a POST request to `api.zebra.com/v2/devices/environmental-sensors/event/subscription` with a body like the following example. Be sure to use your own Client Key, tenant, webhookUrl and the name of your subscription.

The screenshot shows a REST client interface with the following details:

- Method:** POST
- URL:** `https://stage-api.zebra.com/v2/devices/environmental-sensors/event/subscription`
- Body Type:** JSON
- Body Content:**

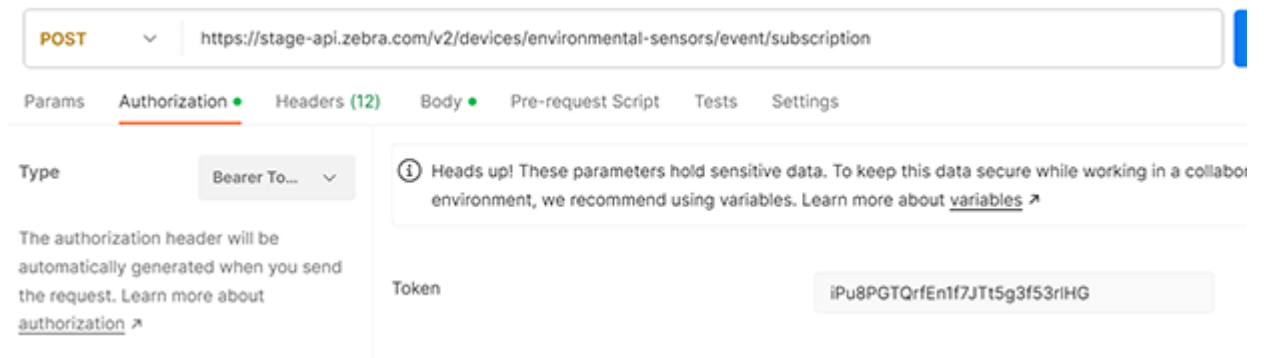
```

1  {
2    "taskIds": [
3      "5daf05cd-7cf0-49f7-a9f6-1b08263a7fa2"
4    ],
5    "epcis": false,
6    "name": "Webhook for task 5daf05cd-7cf0-49f7-a9f6-1b08263a7fa2",
7    "webhookUrl": "https://57b5544b67314ad1c24e2efb9779b19e.m.pipedream.net"
8  }
9

```

Using Webhook Subscriptions

3. Your cURL should look similar to the following example.



4. You will receive a 200 OK response similar to the following example. By default, the webhook subscription starts immediately after its creation.

```
{
  "subscription": {
    "name": "My webhook subscription",
    "changefields": [
      "webhookTrigger"
    ],
    "webhookVerb": "POST",
    "webhookUrl": "https://hooks.myorganization.com/services/hooks/myendpoint",
    "id": "Y47005YJD91NVMFD",
    "headers": {
      "apikey": "2HTxMgwBfw20AmOafjOupjDKJklRXW77",
      "tenant": "zebra"
    }
  }
}
```

Using the GET method on the same API endpoint, you can confirm your webhook subscription by performing a GET request to return all your subscriptions.

Starting a Webhook Subscription

As long as you have a functioning webhook subscription, you can start the subscription at any time by taking the following steps.

1. Send a GET request to `api.zebra.com/v2/devices/environmental-sensors/event/subscription/:subscriptionId/start`. You will need to provide the specific subscriptionId you want to stop as a path variable. Be sure to use your own Client Key and tenant for your subscription.

2. Your cURL should look similar to the following example.

Figure 5 cURL to Start a Webhook Subscription Example

```
curl --location -g --request POST 'https://api.zebra.com/v2/devices/environmental-sensors/event/subscription/B5JD07WPJ3BDVXRR/start' \  
  --header 'Content-Type: application/json' \  
  --header 'Accept: application/json' \  
  --header 'apiKey: 223d7f62dbb247acbc6b132b5f59ad9c' \  
  --header 'tenant: 2e18f99ae1644282a7ead607afb7367d'
```

3. If your subscription started successfully, you will receive an empty 200 OK response.
4. If you are trying to start a subscription that is currently running, you will receive a 409 Conflict response because you cannot start a subscription that is currently running.

Stopping a Webhook Subscription

As long as you have a functioning webhook subscription, you can stop the subscription at any time by taking the following steps.

1. Send GET request to `api.zebra.com/v2/devices/environmental-sensors/event/subscription/:subscriptionId/stop` You will need to provide the specific `subscriptionId` you want stopped as a path variable. Be sure to use your own Client Key and tenant for your subscription.
2. Your cURL should look similar to the following example.

Figure 6 cURL to Stop a Webhook Subscription Example

```
curl --location -g --request POST 'https://api.zebra.com/v2/devices/environmental-sensors/event/subscription/B5JD07WPJ3BDVXRR/stop' \  
  --header 'Content-Type: application/json' \  
  --header 'Accept: application/json' \  
  --header 'apiKey: 223d7f62dbb247acbc6b132b5f59ad9c' \  
  --header 'tenant: 2e18f99ae1644282a7ead607afb7367d'
```

3. If your subscription stopped successfully, you will receive an empty 200 OK response.
4. If you are trying to stop a subscription that is currently stopped, you will receive a 409 Conflict response because you cannot stop a subscription that is not running.

Understanding Webhook Outputs

Zebra can output webhooks in two data formats.

Standard Output

This output includes information on the sensor, task, important timestamps, temperature information, and where the deviation was recorded. The timestamp refers to the timestamp of the event, the recordedTimestamp refers to the timestamp when the data was uploaded.

Figure 7 Standard Webhook Output

```
{
  "event": {
    "id": "6359fcb8-96a0-461c-90b1-07dbb002c063",
    "timestamp": "2023-02-01T07:14:12.893Z",
    "deviceId": "bridge-id_or_phone-id",
    "data": {
      "format": "beacon",
      "id": "sensor_serial_number",
      "value": "-2.06",
      "rssi": -51
    }
  },
  "analytics": {
    "recordedTimestamp": "2023-02-01T07:14:12.893Z",
    "resourceId": "sensor_Task_Id",
    "tenant": "my_tenant",
    "timestamp": "2023-02-01T07:14:12.893Z",
    "meta": {
      "data": {
        "taskId": "task id"
      }
    }
  },
  "enrichment": [],
  "coordinates": {
    "global": {
      "lat": 0,
      "lng": 0
    }
  }
},
"decode": {
  "temperature": {
    "alert": true,
    "deviation": 0.3,
    "format": "celsius",
    "taskId": "task id",
    "sample": -2.06
  }
}
}
```

EPCIS Output

This option displays the output in the GS1 [EPCIS 2.0](#) format and allows the user to integrate temperature alarms into the EPCIS workflows.

Figure 8 EPCIS 2.0 Webhook Output

```
{
  "@context": [
    "https://ref.gs1.org/standards/epcis/2.0.0/epcis-context.jsonld"
  ],
  "type": "EPCISDocument",
  "schemaVersion": "2.0",
  "creationDate": "2023-02-14T21:57:14.339Z",
  "epcisBody": {
    "eventList": [
      {
        "type": "ObjectEvent",
        "eventTime": "2023-02-14T21:56:15Z",
        "eventTimeZoneOffset": "+00:00",
        "epcList": [
          ""
        ],
        "action": "OBSERVE",
        "bizStep": "sensor_reporting",
        "readPoint": {
          "id": "GWJ215000616933F425007C9260C012349132C96B618DE"
        },
        "sensorElementList": [
          {
            "sensorMetadata": {
              "time": "2023-02-14T21:56:15Z",
              "deviceID": "86d5f890-2d09-47dd-ac37-4586cb90ffba",
              "ext1:missionId": "task id",
              "ext1:coordinates": {
                "global": {
                  "lat": 34.149952,
                  "lng": -118.79551
                }
              },
              "ext1:deviation": 3.75,
              "ext1:rssi": -33
            },
            "sensorReport": [
              {
                "type": "Temperature",
                "value": 23.75,
                "uom": "CEL",
                "exception": "ALARM_CONDITION",
                "booleanValue": true
              }
            ]
          }
        ]
      }
    ]
  }
}
```

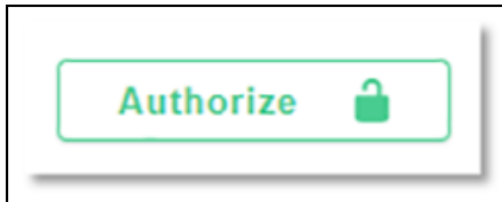
Displaying Full Event Record for Task ID

This use case enables developers to create detailed reports based on the full set of data from a task. The API for this is the [Data Reporting for Electronic Temperature Sensors](#). You first need to get an authorization token using one of the authorization methods previously detailed in this guide.

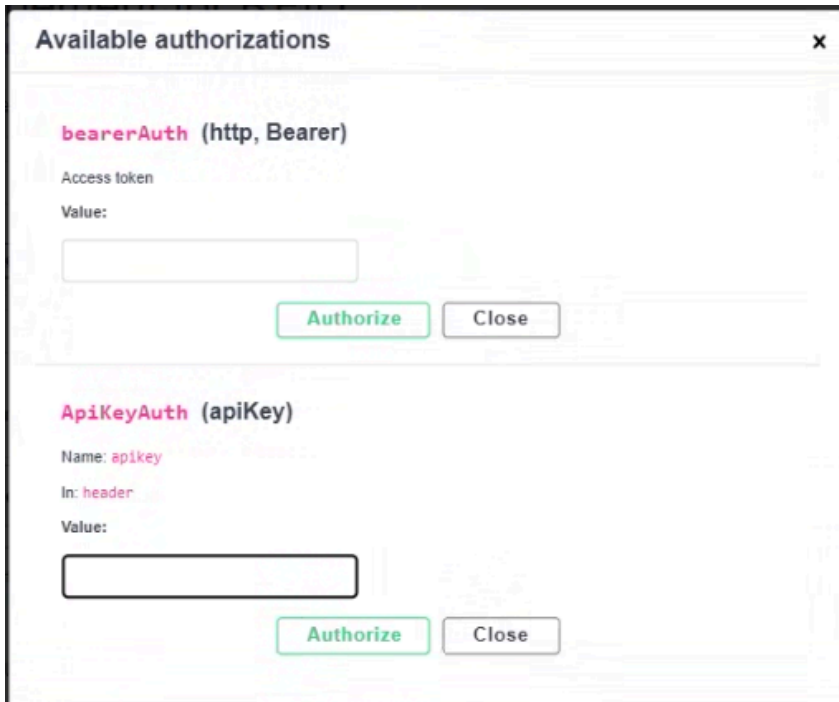
Using the Developer Portal Interactive Documentation

The following steps describe how to use the Zebra Developer Portal interactive documentation.

1. Get your Client Key from the [App page](#). See [Authentication-Simple Key](#) for details.
2. Go to the Documentation page on the Developer Portal at developer.zebra.com/apis/analytics-and-reporting-temperature.
3. Click **Authorize**.



4. Paste the Client Key into the field labeled ApiDeyAuth and click **Authorize**.

A screenshot of a web interface titled "Available authorizations" with a close button (x) in the top right corner. The interface shows two authorization methods: "bearerAuth (http, Bearer)" and "ApiKeyAuth (apiKey)". Under "bearerAuth", there is a label "Access token", a "Value:" label, and an empty text input field. Below the input field are "Authorize" and "Close" buttons. Under "ApiKeyAuth", there is a "Name: apikey" label, an "In: header" label, a "Value:" label, and an empty text input field. Below the input field are "Authorize" and "Close" buttons.

5. Select the /tasks/{taskId}/log endpoint to drop down the documentation.
6. Click **Try It Out**.
7. Enter a Task ID.
8. Click **Execute**.
9. Verify the data response is similar to the response below.

```
{
  [
    "type": "beacon",
    "event": {
      "id": "6359fcb8-96a0-461c-90b1-07dbb002c063",
      "timestamp": 1633359112806,
      "deviceId": "bridge-id_or_phone-id",
      "data": {
        "format": "beacon",
        "id": "sensormac",
        "value": "-2.06",
        "rssi": -51
      },
    },
    "analytics": {
      "recordedTimestamp": 1633359112806,
      "resourceId": "bridge-id_or_phone-id",
      "tenant": "my_tenant",
      "timestamp": 1633359112806,
      "meta": "{\"taskId\":\"067e2b5e-0f70-4010-8245-28361008cca4\"}"
    },
    "decode": {
      "temperature": {
        "alert": true,
        "deviation": 0.3,
        "format": "celsius",
        "taskId": "task id",
        "sample": -2.06
      }
    }
  ]
}
```

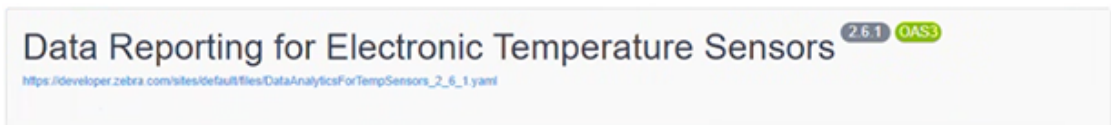
Sorting, Filtering, and Pagination

The read events log services use token-based pagination due to the potentially very large number of results. The default page size is 100. You request the first page of content and optionally request a size (number of sense events) to return. Responses also include a token for the next page. Use this token in the next request, optionally with a size, to get the next set of event data.

Using an API Test Tool

The following steps describe how to use the API test tool.

1. Go to your favorite API test tool. The example below uses Postman, but other tools can also be used.
2. Download the Yaml file from the [Developer Portal](#).



3. Import the Yaml file as a test suite.

Import X

Select files to import - 1/1 selected

NAME	FORMAT	IMPORT AS
Analytics and Reporting for Temperature Se...	OpenAPI 3.0	API

Generate collection from imported APIs

Link this collection as

Test Suite v

[> Show advanced settings](#)

Cancel
Import

Displaying Full Event Record for Task ID

4. Modify the taskID to be a valid task you created. Deselect **startTime** and **endTime**.

The screenshot shows the Postman interface for a GET request. The URL is `{{baseUrl}}/tasks/:taskId/sense-events`. The 'Query Params' section contains a table with the following data:

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input type="checkbox"/> startTime	2022-10-12T08:00:00.006Z	Date-time timestamp of records newe...		
<input type="checkbox"/> endTime	2022-11-08T16:00:00.006Z	Date-time timestamp of records not o...		
Key	Value	Description		

The 'Path Variables' section contains a table with the following data:

KEY	VALUE	DESCRIPTION	...	Bulk Edit
taskId	223d7f62-dbb2-47ac-bc6b-132b5f5...	(Required) The ID number of the task		

5. Go to the Headers tab and enter the Key **apikey** and Value your **apikey** from step 1.
6. Click **Send**.
7. Verify the response is the same as the response you received in step 0.
8. Modify the **startTime** and **endTime** dates as desired to filter the results within a time range.
9. In Postman, click **Code Menu** and select the language for your application.
10. Copy and paste the code snippet into your application code for complete integration.

```
Code snippet
cURL
1 curl --location 'https://api.zebra.com/v2/data/
  environmental/tasks/
  223d7f62-dbb2-47ac-bc6b-132b5f59ad9c/
  sense-events' \
2 --header 'Accept: application/json' \
3 --header 'apikey: 1234'
```

Downloading AIDL Files

The ZS300 ZSFinder app supports the Android Interface Definition Language (AIDL). This enables remote procedure calls between the ZSFinder app and other Android applications.

AIDL defines an interface that details the methods available to be remotely called. Clients bind to the interface to enable communication. Once the client is bound to the interface, it can then make calls to the ZSFinder app. Those call are executed by the ZSFinder app, resulting in it either taking an action or returning requested data, such as sensor type, battery level, data logs, or alarm status.

To get started:

1. Go to the [ZS300 Support Page](#) to download the AIDL files. These include the Javadoc files that contain complete documentation on the supported Classes and Methods.
2. Copy the entire AIDL package directory structure and AIDL files into the `app/src/main/aidl/` directory of the client application. Initiate an Android build so the class files can be generated. The classes will not be usable until the application has been built.
3. Add the following query entry to the `AndroidManifest.xml` inside the `<manifest></manifest>`:

```
<queries> <package android:name="com.zebra.zs300SensorDiscoveryService" />
</queries>
```

The Android device must be running the ZSFinder APK with the required permissions in order for the AIDL interface to function properly.



NOTE: AIDL interface calls should be made on a non-UI thread so that they do not impact app performance or functionality.

Additional information on AIDL is available at developer.android.com/guide/components/aidl

