INEIGHT

Starting Guide for Developers

InEight Integrations

PLATFORM

Changelog

This changelog contains only significant or other notable changes to the document revision. Editorial or minor changes that do not affect the context of the document are not included in the changelog.

Rev	Date	Description
1.0	05-MAR-2019	Initial Release
		Updated template. Added information for creating bearer tokens.
2.0	01-AUG-2019	Published revision.
		Added information about asynchronous GET pattern
		Added note about SourceSystemId updates requiring a support request
3.0	30-OCT-2019	Published revision.
	29-JAN-2020	Updated information about using GET requests to include specific cases for no data found, less than 1kb data, and more than 1kb data.
	25-FEB-2020	Added information about Core GET APIs enhancement to include additional record count data in the response.
	11-MAY-2020	Added Examples on max record counts and skipping.
	22-MAY-2020	Added NOTE to AppLog Field Definitions about all integration error messages returned.
4.0	12-AUG-2020	Formatting changes.
5.0	01-SEP-2021	Content review. Made formatting and grammatical changes.
6.0	27-JUN-2022	Added table of API responses in GET Request and Responses.
7.0	20-JUN-2023	Added information to clarify POST and PATCH function in Updating an Existing Record.
8.0	30-MAY-2025	Added multiple sections, such as Sample HTTP response, API response codes, and Location URL response codes. Deleted sections that were unnecessary or not applicable.
8.1	05-JUN-2025	Updated Additional Troubleshooting. Updated template.

Contents

About This Document

This document is the starting point for developers creating integrations with InEight. This document contains information about InEight APIs, where to find them, how to use them, what security and authentication measures are taken, and other general information that is common to all cloud application APIs.

This document does not cover specific details of each API, such as data elements, structure, prerequisites, data flows, or specific error codes. For those details you will need to reference the InEight Integration Catalog and identify the individual specification documents that cover the specific APIs.

General Information About InEight Integrations

Business Rules for Integrations

While integrations for InEight are designed to be agnostic of who or what system will be using them from outside of InEight, many of them are dependent on specific business requirements for the flow of data. For example, data exposure about time worked for employees and equipment only occurs when specific activities are performed in the UI that indicate the data for those records has reached a specific state. It is important to note here because each integration might require attention to when and how it can be used. Any pertinent information regarding how, when, and what rules apply to the use of an integration are covered in the integration's specification document.

Types of Integrations

InEight integrations are comprised of either of the following:

- Asynchronous REST APIs
  - Primarily used to push data to InEight
  - Some Get APIs are exposed
  - All data is provided in JSON data packages
  - All are managed through our API Management Portal
- Web Services
  - Primarily used to push data to external systems or initiate a request from external systems based on specific user actions in InEight applications
  - All data is expected or provided in JSON data packages
  - Are configured in the InEight UI

Integration Frequencies

InEight supports integrations at any interval required by a customer or third-party application. However, the actual frequency of an integration should be determined by business need and amount of data being transmitted at any one time. For example, updates to Employee records could be sent to InEight in near real-time based on events such as hiring or termination that occur in an ERP or HCM system because these are infrequent and will update a very small set of records at a time. Updates to cost items for actuals, forecast, or budget have a much larger payload (potentially tens of thousands of records) and can result in better performance if it occurs either on demand by a user in InEight or at scheduled off-hour intervals.

When creating, or using integrations for InEight, discussion of the individual integrations, payloads, and frequencies must occur to determine the best impact to all systems.

Data Transformation

InEight generally does not perform transformation or mapping of any data due to the number of unique cases and needs across different customers. Data provided to external systems from InEight will be representative of the raw data in the InEight's business operations. It is generally the responsibility of the customer receiving systems, or a middle tier of an integration to handle mapping and transformation logic.

JSON to Other File Formats

InEight integrations provide all data through externally (to InEight) accessible APIs that produce records in a JSON format. Transformation of records from JSON to other formats (*e.g.,* XML, CSV, etc.) must be handled by the customer via a business logic application such as SAP Process Integrator, Azure Logic Apps, Microsoft BizTalk, Dell Boomi, or other proprietary logic/code.

Data Mapping

Transformation of data in a field from one context to another (*e.g.,* a set of statuses that must be converted to a true/false flag) and mapping of customer data to an integration field between a customer system and the output from InEight APIs or expected input to InEight integrations must be handled by the customer's integration system or middleware.

Business Logic

Business logic required to conform data, fill fields, or evaluate specific data conditions must be handled by the customer. Some examples of business logic are as follows:

- Providing default data when a field from InEight is left blank.
- Evaluation of one or more data fields to determine an expected end result. For example, if an employee has an active status AND does not have hours, set an "at work" field to *false*.

Record Identification

InEight takes into consideration the "system of record" for all stored data. When the data originates from sources that are external to the InEight cloud platform, a unique identifier is required to be provided by the external system for each record to ensure a common value can be used when exchanging, creating, or updating records through integrations. InEight integrations commonly refer to this field as "SourceSystemId".

When managing "Master Data" (data that is normally static and used to classify or provide dimensional data to transactions), it is standard practice for each InEight entity to contain the following types of fields:

- SourceSystemId: The unique identifier that will be used to match records exchanged in integrations. This field is not normally displayed in the InEight UI.
- Natural Key: This is also a unique identifier, but one that is recognizable by users and is always displayed in InEight UIs. In InEight integrations, this field might be labeled as "DisplayId", or "Name", or have a unique label to accommodate specific usability concerns (e.g., Email is used as the natural key for Users).

InEight separates the concept of record identification into these two fields to allow the possibility of using both a system generated identifier such as a GUID as the SourceSystemId to make system-to-system interactions easier, while also using a human recognizable unique value for all business processes handled through user interaction. In cases where an external system uses a natural key as the system-identifiable unique value, it is acceptable to set the SourceSystemId and Natural Key to the same value.

NOTES:	SourceSystemId and Natural Key field types should be considered required whenever they are present in an integration, even if the fields are not marked as technically required for the integration to work.

SourceSystemId is considered a permanent identification method. To avoid the possibility of user error occurrences by staff who might not understand the significance of the field, it is not displayed in the application UI. Once a SourceSystemId is set for a record, it can only be changed through a support request to InEight or using Users_UpdateSourceSystemID API.

The Users_UpdateSourceSystemID API allows users to edit and change SSIDs in one or multiple records using External System programmatically or through APIM manually. This API impacts Users_Import and Users_Get only. There is no restriction on the number of SSIDs that can be updated through this API. We have not put any validation in place to ensure that any user cannot update their own SSID through this API. Quite simply, the reason for this is that the SSID update happens at the service account level and is agnostic of individual persons... This is further explained below:

The External System integrations use a SVC Account (ClientId/Secret) for all transactions (import/GET). So, irrespective of the user (person) or automated system triggering the import, they are processed in the context of a svc account. Hence, the CreatedById and ModifiedById audit columns in the database tables will have "1" (System Admin) if the entities are imported from External Systems.

Master Data Example of Record Identification

The following example shows the JSON format for the Master Data entity "Cost Centers". In this example, there is a specific field called "SourceSystemId" and a field called "CostCenterDisplayId" that represents the Natural Key that will be used in UIs.

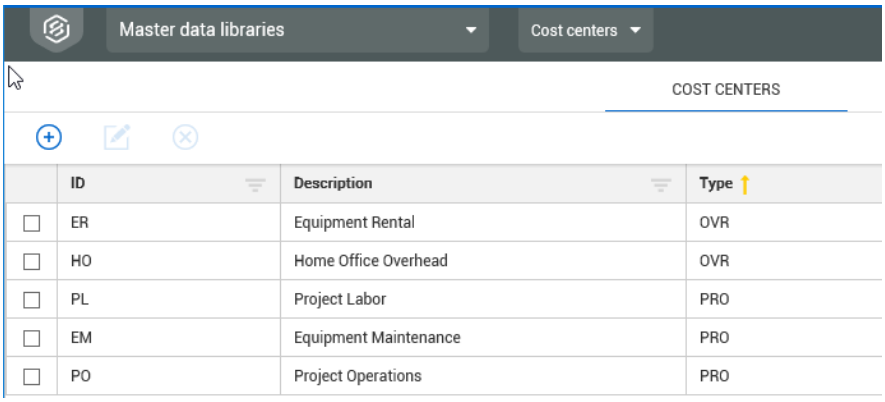
```
[
   {
      "CostCenterDisplayId": "string",
      "CostCenterDescription": "string",
      "CostCenterTypeDisplayId": "string",
      "IsActive": true,
      "SourceSystemId": "string",
      "SourceSystemName": "string"
   }
]
```

When the records were created the API request could have contained several records at once and appeared as follows.

```
[
   {
      "CostCenterDisplayId": "ER",
      "CostCenterDescription": "Equipment Rental",
      "CostCenterTypeDisplayId": "OVR",
      "IsActive": true,
      "SourceSystemId": "OVRER",
      "SourceSystemName": "SAP"
   },
   {
      "CostCenterDisplayId": "HO",
      "CostCenterDescription": "Home Office Overhead",
      "CostCenterTypeDisplayId": "OVR",
      "IsActive": true,
      "SourceSystemId": "OVRHO",
      "SourceSystemName": "SAP"
   },
   {
      "CostCenterDisplayId": "PL",
      "CostCenterDescription": "Project Labor",
      "CostCenterTypeDisplayId": "PRO",
      "IsActive": true,
      "SourceSystemId": "PROPL",
      "SourceSystemName": "SAP"
   },
   {
      "CostCenterDisplayId": "EM",
      "CostCenterDescription": "Equipment Maintenance",
      "CostCenterTypeDisplayId": "PRO",
      "IsActive": true,
      "SourceSystemId": "PROEM",
      "SourceSystemName": "SAP"
   },
   {
      "CostCenterDisplayId": "PO",
      "CostCenterDescription": "Project Operations",
```

```
        "CostCenterTypeDisplayId": "PRO",
        "IsActive": true,
        "SourceSystemId": "PROPO",
        "SourceSystemName": "SAP"
    }
]
```

When the integration is processed, records appear in the UI as follows. Notice that SourceSystemId is not shown, and the CostCenterDisplayId field is renamed to ID in the UI.



If a record needs to be updated, the original SourceSystemId of the record will need to be provided as reference. The below JSON shows the "Project Operations" Cost Center being deleted as an option from Cost Centers.

```
[
    {
        "CostCenterDisplayId": "PO",
        "CostCenterDescription": "Project Operations",
        "CostCenterTypeDisplayId": "PRO",
        "IsActive": false,
        "SourceSystemId": "PROPO",
        "SourceSystemName": "SAP"
    }
]
```

Associated Entity Example of Record Identification

Several InEight integrations represent the union of two or more entities to create an association of master data. The following example uses the Project Craft integration which allows customers to determine which Craft records will be allowed for use on a specific Project.

In the ProjectCraft JSON, "ProjectId" is the "SourceSystemId" of the Project entity, and "CraftId" is the SourceSystemId of the Craft entity.

```
[
   {
      "ProjectId": "string",
      "CraftId": "string",
      "StraightTimeRate": 0.0,
      "OverTimeFactor": 0.0,
      "OverTimeRate": 0.0,
      "DoubleTimeFactor": 0.0,
      "DoubleTimeRate": 0.0,
      "IsActive": true,
      "UseBaseWageFactors": true
   }
]
```

Source System

Each integration contains fields labeled SourceSystemId and SourceSystemName, which is meant to help InEight understand the general source of the data being received. No actions are taken with this information, but it can be useful at a later point to help determine specific system behaviors attached to the incoming data. Work with the InEight Implementations or Professional Services team to establish a unique value that should be contained in these fields for your specific implementations of InEight integrations.

URLs for APIM Calls

When sending integration request messages for InEight APIs, the following URL convention is used. Specific URL addresses for each API can be found in the API's details in APIM.

https://api.ineight.com/integrations/{version}/{api name}/{method | GET parameters}

Migrating to APIM

For the InEight customers on older versions that are currently making direct calls to APIs within their environments, they will need to migrate to APIM. To migrate to APIM, do the following:

1. Sign up for a subscription in APIM
2. Subscribe to the External Integrations product to obtain a subscription key, see API Request Headers for instructions.
3. Point integration calls to APIM endpoints, such as htttps://api.ineight.com/core/employees.
4. Modify message headers to include new required fields per API Request Headers.
    - The authorization header should be the same AAD bearer token that is currently being used.

API Request Headers

All InEight APIs require specific information to be contained within the header of the request message. The following table provides information about the headers.

Header	Required	Type	Description
X-IN8-TENANT-PREFIX	Yes	String	This determines how to route the incoming message to the appropriate Account for authentication and processing. Use the prefix of your InEight environment URL. For example, if your environment URL is https://sample-domain.hds.ineight.com, then the Prefix is "sample-domain".
Content-Type	No	String	Media type of the body sent to the API. Only required on requests that contain content. Use standard MIME types. If POST, "application/json"
Ocp-Apim-Subscription-Key	Yes	String	Subscription key which provides access to this API. The value for this field can be found in your APIM Profile.
Authorization	Yes	String	The Bearer Token created for your authorized user or application. See the section titled "How to Generate a Bearer Token" for more detail.

NOTE: While never recommended by InEight for InEight applications, unmanaged (non-APIM) API requests only require the Authorization header.

API Authentication and Authorization

API requests made to InEight require two authorizations.

Ocp-Apim-Subscription-Key: The first is your authorization to use APIM, which is handled by creating an account in InEight's APIM portal, subscribing the account to Products in APIM, and then passing the generated Subscription Key in the "Ocp-Apim-Subscription-Key" header of the API request. This is described in more detail in the "Generating an APIM Subscription Key" section of this document.

Authorization: The second authorization is done in two parts. Part one is authentication of an Active Directory account associated to a User in InEight. Part 2 is authorization of that User to perform actions in InEight. Both parts of this authorization are handled by passing a Bearer Token in the "Authorization" header of an API request. The Bearer Token is used to authenticate an Active Directory account, look up the User associated to that account, and determine if the User has the appropriate permission in InEight.

NOTE: The User associated to the AD account must have the permission "Manage external API's" assigned to them in InEight.

How to Generate a Bearer Token

The Bearer token is acquired by sending an HTTP request as follows:

```
POST /<TenantId>/oauth2/token HTTP/1.1
Host: login.microsoftonline.com
Content-Type: application/x-www-form-urlencoded
cache-control: no-cache
grant_type=client_credentials&client_id=<ClientId>&client_secret=<ClientSecret>&resource=<TargetResource>
```

If the request is successful, the response looks something like this:

```
{
  "token_type": "Bearer",
  "expires_in": "3600",
  "ext_expires_in": "0",
  "expires_on": "1539884258",
  "not_before": "1539880358",
  "resource": "<TargetResource>",
  "access_token": "<AccessToken>"
}
```

For subsequent API requests to InEight, pass the returned <AccessToken> value in the Authorization header. The AccessToken is typically good for one hour. You can determine the exact time the token will expire by taking the value for "expires_on" and adding that number of seconds to Jan 1, 1970 12:00 AM GMT. When that time nears or passes, you should make the request to acquire a new token before making additional API calls to InEight.

Using GET with InEight APIs

Several InEight integrations allow customers to retrieve data using a Get method on the API request. Most of those integrations use an asynchronous pattern for processing the data as outlined below. Any integrations that support a Get method that do not follow the asynchronous pattern will have their processing method described in detail within their integration specification.

GET Request and Responses

When a Get request is received by the InEight API it first validates the request matches the expected JSON payload criteria. If it passes, a response of 202 Accepted is returned to the caller along with an element labeled as Location that provides the URL address where the requested data can be retrieved. Additionally, a Retry-After header indicating the number of seconds client should wait before making another request is returned.

If the requested data is less than 1kb, the location URL will contain the data in JSON format.

Example for Countries_Get using $filter=startswith(Name, 'Be')

{"@odata.context":"https://apitest.ineight.com/integrations/$metadata#Countries","value":[{"ISOCode":"BY","Name":"Belarus","CountryTranslations":null},
{"ISOCode":"BE","Name":"Belgium","CountryTranslations":null},{"ISOCode":"BZ","Name":"Belize","CountryTranslations":null},
{"ISOCode":"BJ","Name":"Benin","CountryTranslations":null},{"ISOCode":"BM","Name":"Bermuda","CountryTranslations":null}]}

- If the requested data exceeds 1kb, a file containing the requested data in a JSON format will be placed in the URL.
- If no data was able to be found based on filter criteria ($filter) added to the request, the location will contain a message stating: "No Result Values".

```
{"@odata.context":"https://apitest.ineight.com/integrations/$metadata#Countries","value":"No Result Values"}
```

The actual processing of the request and any additional OData or parameterized query information is then handed off to the business logic of the InEight application that owns the data entities involved. It can take anywhere from milliseconds to a couple of minutes for the application to fulfill the request and place the data file in the expected "Location".

During this processing time, customer systems poll the "Location" URL every {Retry-After} seconds to see if the file is available.

- If it is not available, the poll request will return a response of "202".
- Once available, the response will change to "200" and the file will be included.

InEight API endpoints return a 202 Accepted response rather than a simple 200 OK response. The 202 Accepted response includes a **Location** header with a URL that customers can call periodically, with an interval of seconds specified in Retry-After header, which will return either another 202 Accepted response indicating that the import is still in progress, a 200 OK response with an optional result payload indicating the import completed successfully/partially, or a 4XX/5XX response indicating that the import failed or timed out.

Sample HTTP response

HTTP/1.1 202 Accepted
Cache-Control: no-store, must-revalidate, no-cache
Pragma: no-cache
Content-Length: 0
Location: https://{tenant-prefix}.hds.ineight.com/ExternalSystem//Messages/ImportStatus?requestId=6ede5356-cbd8-4602-81b1-e4150bc1c2f1&messageId=1a47aa63-59c8-40fe-b2db-43e43b37744f
Retry-After: 10
Access-Control-Expose-Headers: Request-Context
Strict-Transport-Security: max-age=31536000; includeSubDomains; preload
X-RequestID: 8ade2530-bf48-446d-8648-a9be45fdf961
Request-Context: appId=cid-v1:ed9ea0ab-37bd-4c25-acd6-d6cb56bbc8d8
X-Powered-By: ASP.NET

Date: Thu, 29 May 2025 20:09:50 GMT

API response codes

The API could return the following responses.

Code	Description	Details
200	Response	Response code when a request returns synchronously with a response. *This response code is rarely seen because most InEight APIs are asynchronous. Synchronous APIs will be called out individually in the API specific documentation*
202	Request accepted, queued, or processing	Response code when a request that has been accepted for processing.
400	Bad request	Response code when payload is not complete. An example would be a payload is missing required fields.
401	Unauthorized	Response code when authorization type not specified. An example would be an authorization flow not specified in the request.
403	Forbidden error	Response code when your authorization method doesn't have permission to access the external API. The permission required for external APIs is under Suite administration -> Application integrations -> "Manage external API's".
404	Invalid end point	Response code when the URL provided is invalid. If the Location header URL is accessed after it's expired, the status will be a 404.
405	Method not allowed	Response code when using an HTTP method (like POST, GET, PUT, DELETE) that the resource doesn't support. An example would be using PATCH on a POST operation.
417	Except header not allowed	Response code when the API doesn't support the Except request in the header.
500	Server Error	Response code when there's an internal server error. An example would be if an invalid tenant prefix provided.
502	Bad gateway	Response code when there's an incorrect response from within the communication on InEight's systems.
503	Service unavailable	Response code when the server is temporarily unable to handle the request. An example would be if a deployment is occurring, and the Azure services are down.

Location URL response codes

The Location/Status header endpoint (https://{tenant-prefix}.hds.ineight.com/externalsystem/messages/status?messageId={messageId})

Response URL Security

An environment configuration is available to increase the security around the authentication of API response URL messages. If the environment setting is configured on, the time-out of a response URL will be set to 15 minutes. The consumer will have to provide a token with the response URL to access the response message.

If the Response URL security is configured, you will have to provide the below header value to access the Response URL.

Header	Required	Type	Description
Authorization	Yes	String	The Bearer Token created for your authorized user or application. See the section titled "How to Generate a Bearer Token" for more detail.

Response codes

Code	Description	Details

200	Response	• For GET requests:
		○ If the payload is less than 1KB, it is returned as the content/body of the response.
		○ If the payload is greater than 1KB, then the payload is downloaded as a json file.
		• For Import requests:
		○ If all the records were successfully imported, the response content/body will be:
		{"Logs":]}
		If it is a partially succeeded request, the response will have the validation errors as:
		{
		"Logs": [
		"Ignoring Trade having display '{TradeId}' as it is having duplicate SourceSystemId or DisplayId",
		"Imported 0 of 1 Trade entities: Adds(0) Updates(0) Deletes(0) Errors(0) Skipped(1)"
		]
		}

202	Request Processing	Request is still processing.
302	Redirect blob response URL	If additional response security is not enabled, you could receive a redirect blob response URL.
404	Not found	Response URL is no longer available, or an invalid URL has been entered
500	Server Error	Response code when the process failed with an exception.

Additional Troubleshooting

Errors are logged within your environment. Go to the following URL to access the additional error log information.

https://{tenant}.hds.ineight.com/applogs

Data Availability

InEight APIs that follow this pattern make all the data for the requested entity available at any time. For instance, a request can be made at any time to retrieve every Employee or Equipment resource available in the customer's InEight environment. However, to prevent an unnecessarily large request from processing each time, some APIs require or optionally ask for specific query parameters to filter the data to a smaller subset.

NOTE:	InEight APIs that support a GET method, but do not follow this pattern, will have their specific processing method described in their integration specification. For instance, the Daily Plans API works by adding Daily Plan records to an integration queue each time one is approved, and the GET request will return only records that exist in the queue at the time of the request.

Maximum Record Counts

To prevent accidental overloading of system resources, GET requests that follow this pattern will return a maximum number of records each time the request is made. To return all available records for an entity, it is up to the customer's systems to make as many requests as necessary (using the OData $skip parameter) until all records are received.

NOTE:	Each integration that supports a GET method will have its maximum record count stated either in the integration specification, or in the description of the API in APIM.

Record Counts in Core GET APIs

All APIs provided by the Core application that support a GET method include additional information about record counts when the OData $count parameter is included and set to "true" in the request. This additional information should help developers using the Core APIs to determine how many records they should expect to receive based on their request parameters and determine how many times they might need to make subsequent requests using $skip before all records are received.

The additional record count information includes:

- **@odata.count** – The count of all records in the entity
- **QueryCount** – The total number of records that should be returned from the entity based on any OData filters or other criteria added to the request.
- **StartingRecord** – Based on the QueryCount and any "$skip" request, determines which record number is represented as the first record within the response data.
- **EndingRecord** – Based on the QueryCount and any "$skip" request, determines which record number is represented as the last record within the response data.

Examples

Example 1 - No specific $filter criteria

- Customer requests data using the "Regions_GET" API.
- Core contains 3,531 records for regions (returned as "@odata.count")
- Core provides the first 1,000 records to the requester as the returned data set
- The JSON payload contains the following information:
    - "@odata.count": 3531
    - "QueryCount": 3531
    - "StartingRecord": 1
    - "EndingRecord": 1000

The customer now knows that there are 3,531 total records to retrieve, and they have 1,000 of them so far...

- Customer makes another request using "Regions_GET", but specifies a $skip value of 1000
- Core provides the next 1,000 records as the returned data set
- The JSON payload contains the following information:
    - "@odata.count": 3531
    - "QueryCount": 3531
    - "StartingRecord": 1001
    - "EndingRecord": 2000

The customer continues to make requests using $skip until the EndingRecord reaches 3531

Example 2 - Customer uses $filter criteria

- Customer requests data using the "Regions_GET" API with $filter=startswith(CountryISOCode, "C")
- Core contains 3,531 records for regions (returned as "@odata.count")
- After applying the filter there are still 1,212 records that meet the request of the customer
- Core provides the first 1,000 records to the requester as the returned data set
- The JSON payload contains the following information:
    - "@odata.count": 3531
    - "QueryCount": 1212
    - "StartingRecord": 1
    - "EndingRecord": 1000

The customer now knows that there are 1,212 total records to retrieve, and they have 1,000 of them so far...

- Customer makes another request using "Regions_GET" with the same $filter criteria and specifies a $skip value of 1000
- Core provides the next 1,000 records as the returned data set
- The JSON payload contains the following information:
    - "@odata.count": 3531
    - "QueryCount": 1212
    - "StartingRecord": 1001
    - "EndingRecord": 1212

The customer does not need to make any more requests because the EndingRecord now matches the QueryCount.

The following example show the record counts after setting the $count parameter to true and setting the $top parameter to 1.


```
{
  "@odata.context":"https://apitest.ineight.com/integrations/$metadata#Trades
",
  "value":[
   {
     "TradeDisplay":"AD",
     "TradeDescription":"",
     "IsActive":true,
     "SourceSystemId":"AD",
     "SourceSystemName":null

}

],
  "@odata.count":853,
  "QueryCount":853,
  "StartingRecord":1,
  "EndingRecord":1
}
```


Using the same data set and setting the $count parameter to true and $skip to 300 results in the following record counts (note that each request returns 500 records for this API):

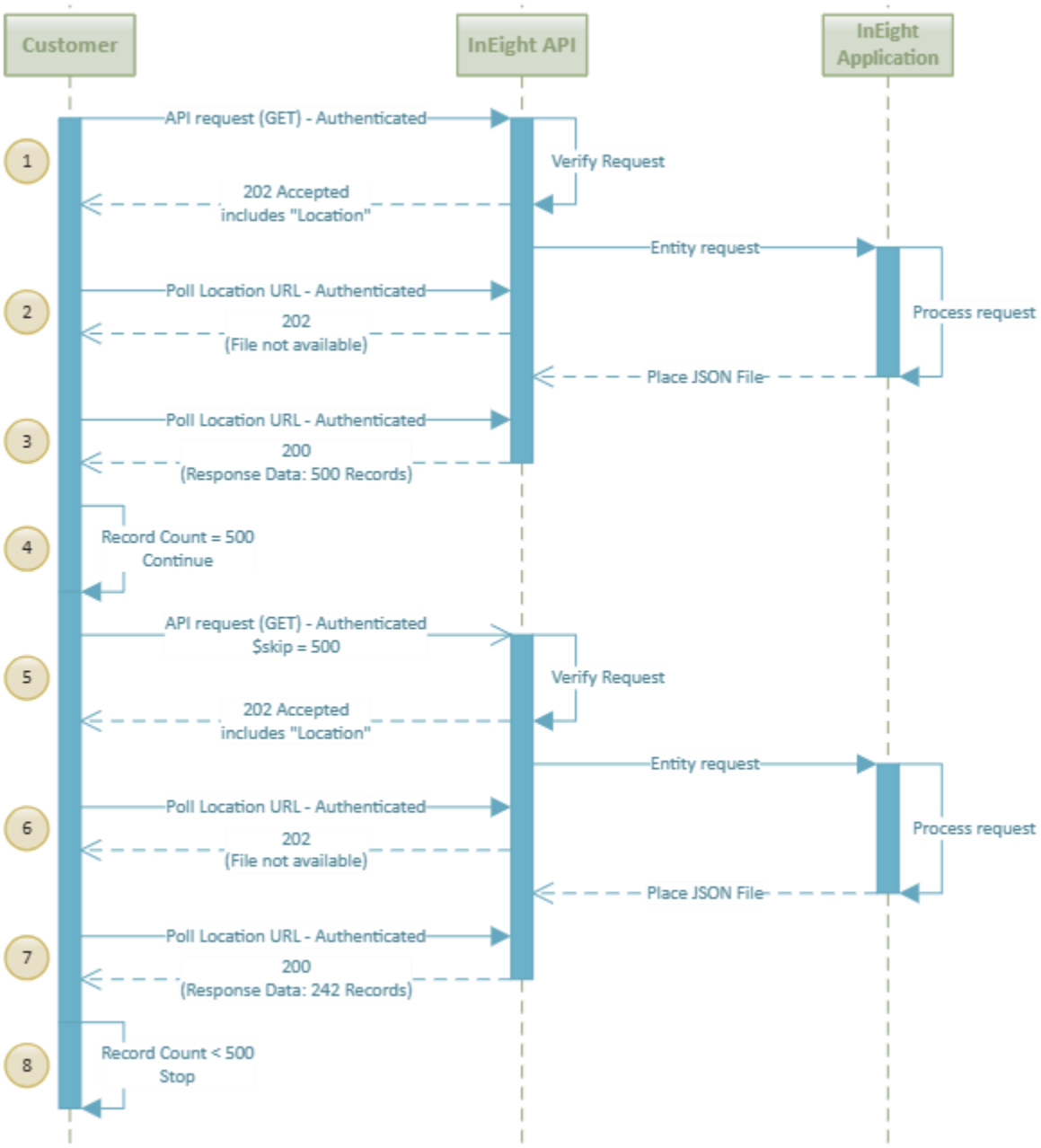

```
],
  "@odata.count":853,
  "QueryCount":853,
  "StartingRecord":301,
  "EndingRecord":800
}
```


Again, using the same data set; setting $count to true and $filter to "startswith(TradeDisplay, 'A')" results in the following record counts:


```
],
  "@odata.count":853,
  "QueryCount":40,
  "StartingRecord":1,
  "EndingRecord":40
}
```

Example Asynchronous GET Process

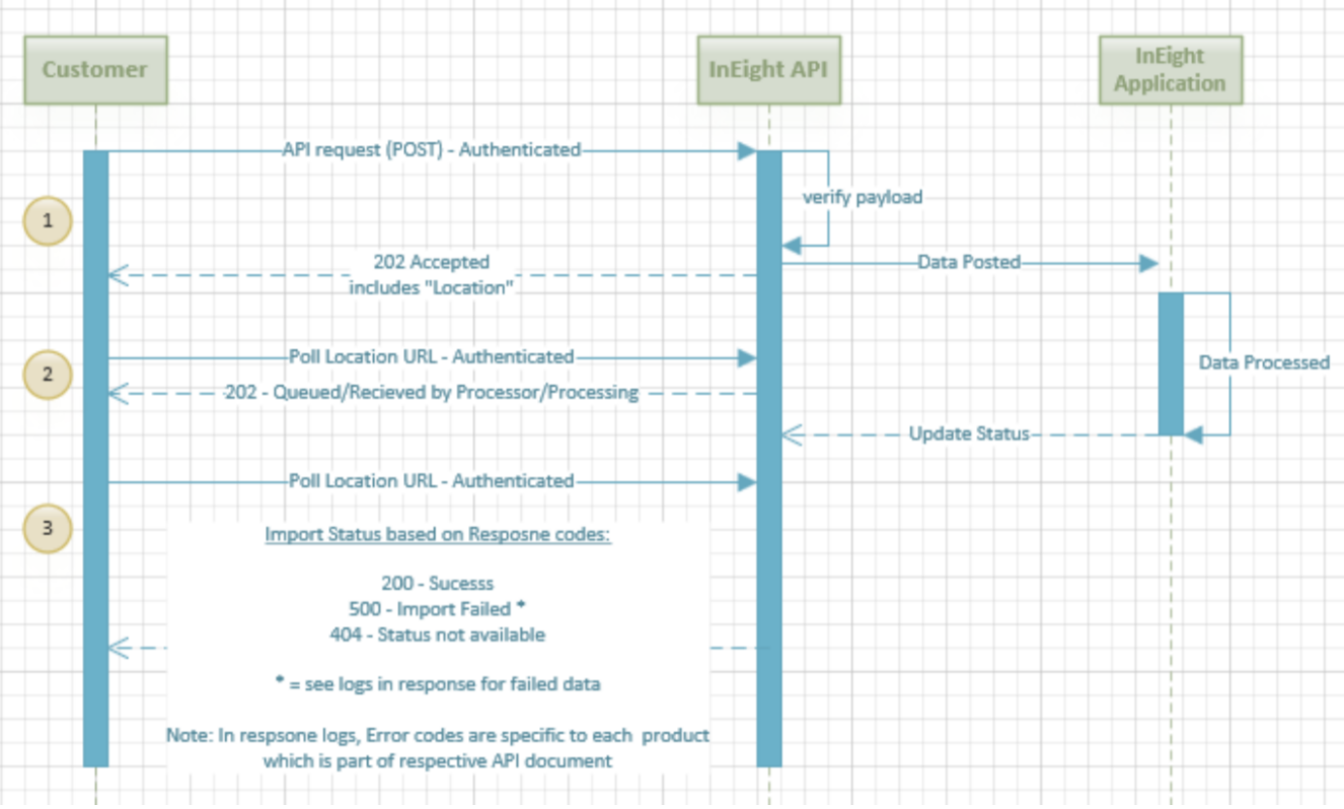
The following example shows the "happy path" that would be followed when retrieving data for an entity and does not include any handling of error conditions. In this example the API returns 500 records at a time and the entity being queried has 742 records.



Process	Description
1	• The customer system makes a GET request to an InEight API. • The InEight API successfully validates the request and returns a "202" response containing the "Location" element. • The request is handed off to the InEight application business logic for processing
2	• The customer system polls the URL provided in the Location element • Because the JSON payload file is not yet ready, InEight returns a "202" response
3	• The customer system continues to poll the URL provided in the Location element • The JSON payload file is not available and InEight returns a "200" response with the file
4	• Because the JSON payload file contains the maximum number of records that can be returned by the API (500), the customer system determines that it should initiate another request to get more records
5	• The customer system makes a GET request to an InEight API and specifies that the first 500 records should be skipped ($skip=500) • The InEight API successfully validates the request and returns a "202" response containing the "Location" element. • The request is handed off to the InEight application business logic for processing
6	• The customer system polls the URL provided in the Location element • Because the JSON payload file is not yet ready, InEight returns a "202" response
7	• The customer system continues to poll the URL provided in the Location element • The JSON payload file is not available and InEight returns a "200" response with the file
8	• This time the JSON payload file contains 242 records, which is below the maximum that can be returned by the API, thus indicating there should be no more records and the customer system can stop making requests.

Example Asynchronous POST Process

The following example shows the flow of using POST/Import APIs.



Process	Description
1	• The customer system makes a POST request to an InEight API. • The InEight API successfully validates the request and returns a "202 Accepted" response containing the "Location" and “Retry-After” headers. • The request is handed off to the InEight application for processing
2	• The customer system continues to poll the URL provided in the Location element after every {Retry-After} seconds. • Because the request is still being processed, the response will still be a “202”.
3	When InEight completes the processing of the request, a "200 OK" response is returned.

Update, Insert, and Delete Operations for Inbound Data

When InEight receives records within an integration, the following actions are performed:

Look for a match on the SourceSystemId field.

a. If a match is found,
   i. If IsActive is set to *true*.
      - Update the matching InEight record with the values provided in the integration record.
   ii. If IsActive is set to *false*.
      - Soft delete the InEight record.
b. If a match is **not** found,
   i. Create a new record in InEight using the values from the integration record.

Creating a New Record

Send a record via the integration that does not have an existing match on the SourceSystemId field, which is identified in each integration specification.

NOTE:	If a new record has a matching natural key for a prior record in InEight that has IsActive set to *false*, and the IsActive field on the new integration record is set to *true*, the system will create a new record instead of reactivating the original record.

Updating an Existing Record

Send a record via the integration that has a match on the SourceSystemId field with the values that have changed for the record in the full payload. When using a POST to update a record, the full payload is expected and therefore if a field is not passed, it is considered empty and is updated accordingly. To update only specific fields in a record, such as for Employees, use a PATCH API.

Deleting a Record

InEight does not permanently delete records from the database. Records are instead flagged as "Inactive" to hide them from system functionality. This allows the records to remain intact to preserve referential integrity where needed.

To remove a record, send the record via the integration that has a match on the SourceSystemId field and set the IsActive field to *false*.

A Note About Updating Data and Data Mismatches

As a general rule for all applications in InEight, the method for handling data race conditions is that the last update made against a record is always applied. When an entity (*e.g.,* Employees) is managed via integration, but Users in InEight also have permission to edit the entity directly through the UI, this could cause conditions where data in the source system does not match expected data in InEight due to manual edits of records. When troubleshooting data mismatches please consider the following:

- Carefully review "last updated" time stamps and "last updated by" IDs to see if they match expected values from integrations.
- Verify who has permission to directly edit the data in InEight.
- Ensure that integration requests to update a record within an entity only contain updated values for the fields that should be updated.

Changing Associated Records

There are some cases where the association of a record from one entity to another must be changed, such as when an active employee changes location from one project to another. In this case, the Employee record does not change (Employee Integration), but the association of the employee to a project (Project Employee Integration) does. The exact specifics of what needs to change and how the integration needs to be triggered are determined by the customer. For instance, an employee can move from one project to another different project, or the employee can be assigned to both projects simultaneously. In another case, a Project record can change, and all employees assigned to the project might need to be updated. The following cases provide examples of how to handle the association updates.

Case: An Employee moves from one Project to another

1. Send a ProjectEmployee integration record with the EmployeeDisplay for the impacted Employee, ProjectId for the project that they are moving FROM, and IsActive is set to *false* to remove the employee from the project.
2. Send a ProjectEmployee integration record with the EmployeeDisplay for the impacted Employee, ProjectId for the project that they are moving TO, and IsActive is set to *true* to add the employee to their new project.

Case: Employee assigned to a new Project in addition to existing Project(s)

Send a ProjectEmployee integration record with the EmployeeDisplay for the impacted Employee, ProjectId for the project that they are being assigned to, and IsActive is set to *true* to add the employee to their new Project.

Case: All Employees assigned to a new Project

Follow the same pattern as "Case: Employee changes Projects" for all impacted employees.