SuiteTalk REST Web Services (Beta)
should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle Master Agreement, Oracle License and Services Agreement, Oracle PartnerNetwork Agreement, Oracle distribution agreement, or other license agreement which has been executed by you and Oracle and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced, or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

For information about Oracle’s commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Sample Code

Oracle may provide sample code in SuiteAnswers, the Help Center, User Guides, or elsewhere through help links. All such sample code is provided “as is” and “as available”, for use only with an authorized NetSuite Service account, and is made available as a SuiteCloud Technology subject to the SuiteCloud Terms of Service at www.netsuite.com/tos.

Oracle may modify or remove sample code at any time without notice.

No Excessive Use of the Service

As the Service is a multi-tenant service offering on shared databases, Customer may not use the Service in excess of limits or thresholds that Oracle considers commercially reasonable for the Service. If Oracle reasonably concludes that a Customer’s use is excessive and/or will cause immediate or ongoing performance issues for one or more of Oracle’s other customers, Oracle may slow down or throttle Customer’s excess use until such time that Customer’s use stays within reasonable limits. If Customer’s particular usage pattern requires a higher limit or threshold, then the Customer should procure a subscription to the Service that accommodates a higher limit and/or threshold that more effectively aligns with the Customer’s actual usage pattern.

Beta Features

Oracle may make available to Customer certain features that are labeled “beta” that are not yet generally available. To use such features, Customer acknowledges and agrees that such beta features are subject to the terms and conditions accepted by Customer upon activation of the feature, or in the absence of such terms, subject to the limitations for the feature described in the User Guide and as follows: The beta feature is a prototype or beta version only and is not error or bug free and Customer agrees that it will use the beta feature carefully and will not use it in any way which might result in any loss, corruption or unauthorized access of or to its or any third party’s property or information. Customer must promptly report to Oracle any defects, errors or other problems in beta features to support@netsuite.com or other designated contact for the specific beta feature. Oracle cannot guarantee the continued availability of such beta features and may substantially modify or cease providing such beta features without entitling Customer to any refund, credit, or other compensation. Oracle makes no representations or warranties regarding functionality or use of beta features and Oracle shall have no liability for any lost data, incomplete data, re-run time, inaccurate input, work delay, lost profits or adverse effect on the performance of the Service resulting from the use of beta features. Oracle’s standard service levels, warranties and related commitments regarding the Service shall not apply to beta features and they may not be fully supported by Oracle’s customer support. These limitations and exclusions shall apply until the date that Oracle at its sole option makes a beta feature generally available to its customers and partners as part of the Service without a “beta” label.
Send Us Your Feedback

We'd like to hear your feedback on this document.

Answering the following questions will help us improve our help content:

- Did you find the information you needed? If not, what was missing?
- Did you find any errors?
- Is the information clear?
- Are the examples correct?
- Do you need more examples?
- What did you like most about this document?

Click here to send us your comments. If possible, please provide a page number or section title to identify the content you're describing.

To report software issues, contact NetSuite Customer Support.
# Table of Contents

SuiteTalk REST Web Services Overview and Setup ................................................................. 1  
  REST Web Services Key Concepts .................................................................................. 2  
  REST Web Services and Other Integration Options ...................................................... 2  
  REST Web Services Prerequisites and Setup .................................................................. 3  
  REST Web Services URL Schema and Account-Specific URLs ......................................... 4  
  Authentication ............................................................................................................. 6  
  Working with REST Web Services Using Postman .......................................................... 7  
    Importing and Setting Up a Postman Environment ..................................................... 8  
    Importing the Postman Collection ............................................................................ 9  
    Sending a Request From the Imported Collection ................................................... 9  
Working with Resource Metadata ...................................................................................... 10  
  Getting Metadata ....................................................................................................... 11  
  Working with OpenAPI 3.0 Metadata .......................................................................... 12  
  Working with JSON Schema Metadata ...................................................................... 15  
Working with Records ...................................................................................................... 18  
  NetSuite Record Structure ........................................................................................... 18  
  Creating a Record Instance .......................................................................................... 19  
  Getting a Record Instance ............................................................................................ 19  
    Format of Sublists and Subrecords ......................................................................... 20  
    Format of Selects and References ........................................................................... 21  
    Format of Multiselects .............................................................................................. 21  
    Format of Enumeration Values .............................................................................. 22  
  Updating a Record Instance ......................................................................................... 22  
  Using the Upsert Operation ......................................................................................... 23  
  Deleting a Record Instance .......................................................................................... 23  
  Using External IDs ...................................................................................................... 23  
Working with Sublists ...................................................................................................... 25  
  Creating a Sublist ...................................................................................................... 26  
  Updating a Sublist ...................................................................................................... 27  
  Replacing a Sublist ..................................................................................................... 29  
Working with Subrecords ................................................................................................... 35  
Record Filtering and Query ............................................................................................... 37  
  Listing All Record Instances ..................................................................................... 37  
  Record Collection Filtering ......................................................................................... 38  
  Collection Paging ........................................................................................................ 40  
Error Handling in REST Web Services ............................................................................. 42
SuiteTalk REST Web Services Overview and Setup

**Warning:** REST Web Services is a beta feature. The contents of this feature are preliminary and may be changed or discontinued without prior notice. Any changes may impact the feature’s operation with the NetSuite application. Oracle warranties and product service levels shall not apply to the feature or the impact of the feature on other portions of the NetSuite application. Oracle may review and monitor the performance and use of this feature. The documentation for this feature is also considered a beta version and is subject to revision.

The NetSuite REST web services provide an integration channel that extends the capabilities of SuiteTalk. REST web services provide a REST-based interface for interacting with NetSuite.

Using REST web services beta version, you can:

- Use CRUD (create, read, update, delete) operations to perform business processing on NetSuite records and to navigate dynamically between records. For details, see Working with Records.
- Get and process the API definition and record metadata. For details, see Working with Resource Metadata.
- Execute NetSuite queries on records. For details, see Record Filtering and Query.

**Benefits of REST Web Services**

The main benefits of REST web services include the following:

- Simple access to records metadata. This includes user and company-specific metadata. For more information about working with records metadata, see Working with Resource Metadata.
- Easier handling of custom records and custom fields.
- Easy to navigate API.
- In contrast to RESTlets, you do not need to write, deploy, and run custom scripts.

**Limitations of the Beta Version**

Consider the following limitations when working with the beta version of the REST web services.

- The REST web services beta version does not support external ID and name ID references. When working with REST web services, you must use internal IDs to refer to records.
- Creating, updating, and deleting subrecords is not supported.
- Enumeration values must be provided and are returned in the SuiteScript internal format. This is the format used by the getFieldValue() and setFieldVal() functions. For information about using these methods, see the SuiteScript Developer Guide.
- Query only returns record IDs and HATEOAS links. (See HATEOAS.) That is, query results have a form of non-expanded references. Additionally, you can only use body fields in query conditions.
- Making queries on transactions and certain other record types is not supported.

For more information, see the following topics:
REST Web Services Key Concepts

The following sections introduce the main concepts of REST web services.

HATEOAS

Hypermedia as the Engine of Application State is an essential principle that should be respected in RESTful APIs.

In practice this means that you can navigate to the referenced resources without deeper knowledge of the system. A typical response contains “links” sections for each resource, which can be a sub-resource of a parent resource or any other referenced resource. You can use links to work with those resources.

For example, when getting sales order data, the response contains a customer reference field that contains a links section. You can then use the link to get data of the particular customer.

For more information, see the following resources:

- [https://en.wikipedia.org/wiki/Link_relation](https://en.wikipedia.org/wiki/Link_relation)

Resource

A resource represents some data which can be uniquely identified. Each resource has its own unique URL, and each resource can reference other resources.

The two main types of resources are the following:

- Singular resources
- Collection resources that contain multiple singular resources

Resources can exist in hierarchy, and can form a tree structure, consisting of child and parent resources.

In NetSuite, the most important resource is a record. A record is a singular resource. However, there can be other resources in NetSuite as well.

A record usually references other resources - other records.

An example of a collection resource is a sublist because it contains multiple lines. Each line is a singular child resource, and the record is a parent resource.

A record with multiple sublists, each of them with multiple lines, forms a hierarchical resource.

REST Web Services and Other Integration Options

To decide the best integration option for your purposes, consider the following comparisons.
The following table compares the characteristics of REST web services with those of SOAP web services and RESTlets.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>REST Web Services</th>
<th>SOAP Web Services</th>
<th>RESTlets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported Operations</td>
<td>get, search, add, update, delete</td>
<td>get, search, add, update, delete</td>
<td>get, search, add, update</td>
</tr>
<tr>
<td>Authentication Supported?</td>
<td>Yes (token-based authentication)</td>
<td>Yes (user credentials; token-based authentication)</td>
<td>Yes (user credentials; token-based authentication)</td>
</tr>
<tr>
<td>Passing of Login Details</td>
<td>in OAuth authorization header</td>
<td>in body (SOAP)</td>
<td>in authorization header</td>
</tr>
<tr>
<td>Passing of Parameters</td>
<td>all parameters on URL</td>
<td>all parameters in body (SOAP)</td>
<td>GET parameters on URL</td>
</tr>
<tr>
<td>Supported Content Types</td>
<td>JSON, Swagger</td>
<td>text/xml (explicit)</td>
<td>JSON, text/xml (explicit)</td>
</tr>
<tr>
<td>Environment</td>
<td>lightweight, no coding and script deployment needed on the server side</td>
<td>heavy programming and deployment environment (C#, Java)</td>
<td>lightweight, suitable for mobile devices, bundleable</td>
</tr>
<tr>
<td>Functionality</td>
<td>Metadata catalog. See Working with Resource Metadata</td>
<td>CRUD</td>
<td>All functionality available through SuiteScript:</td>
</tr>
<tr>
<td></td>
<td>CRUD. See Working with Records.</td>
<td>Search</td>
<td>- CRUD</td>
</tr>
<tr>
<td></td>
<td>Record Filtering and Query</td>
<td></td>
<td>- Search</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>- Query</td>
</tr>
<tr>
<td>Standards</td>
<td>Similar to the REST APIs of other Oracle products</td>
<td>SOAP protocol</td>
<td>No standards</td>
</tr>
<tr>
<td>Required User Knowledge</td>
<td>REST programmer API level knowledge</td>
<td>SOAP programmer API level knowledge</td>
<td>JavaScript programmer SuiteScript level knowledge</td>
</tr>
</tbody>
</table>
To assign the necessary permissions to a role:

1. Go to Setup > Users/Roles > User Management > Manage Roles.
2. Locate the role you want to modify. Click the corresponding **Edit** or **Customize** link.
3. On the Permissions subtab, click Setup.
4. In the Permission list, select REST Web Services.
5. In the Level list, select Full.
6. Click Add.
7. On the Permissions subtab, click Reports.
8. In the Permission list, select SuiteAnalytics Workbook.
9. In the Level list, select Full.
10. Click Add.
11. Click Save.

**REST Web Services URL Schema and Account-Specific URLs**

You can only access REST web services using account-specific domains. Account-specific domains are unique to your account because they contain your account ID as part of the domain name. These domains do not change when your account is moved to a different data center.

When using account-specific domains, dynamic domain discovery is not needed.

The format of an account-specific domain name is: `<account ID>.<service>.netsuite.com`. For example, if your account ID is 123456, your account-specific domain for REST web services would be: 123456.suitetalk.api.netsuite.com

**Important:** The examples in this document use DEMO123 as the account ID. Substitute this with your actual account ID.

A URL for a REST resource looks like the following.

<table>
<thead>
<tr>
<th>Example 0: Test Request</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
</tr>
<tr>
<td><code>https://&lt;accountID&gt;.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/select-customer</code></td>
</tr>
</tbody>
</table>

The URL consists of the following parts:

1. Protocol
2. The account ID of your NetSuite account
3. The domain name for REST web services
4. The complete REST web services endpoint
5. The specific API
6. The version of the specific REST web services API
7. The facility (either metadata catalog or record)
8. The resource that is being accessed
9. Query parameters
Authentication

NetSuite supports token-based authentication (TBA), a robust, industry standard-based mechanism that increases overall system security. This authentication mechanism enables client applications to use a token to access NetSuite through APIs, eliminating the need for integrations to store user credentials.

The REST web services beta also uses TBA for user login. There is no support for login through user credentials. For general information about TBA, see the help topic Token-based Authentication (TBA).

Setting Up Token-Based Authentication (TBA)

To set up TBA, see the help topic Getting Started with Token-based Authentication.

To be able to use REST web services with TBA, you must create an application using an integration record. See the help topic Creating Integration Records for Applications to Use TBA. As the last step of this procedure, make sure you note the consumer key and consumer secret.

After creating the integration application, continue with creating tokens for your users. Issue a new token for at least one of your users, and note its token ID and token secret. For details, see the help topic Access Token Management – Create and Assign a TBA Token.

Concurrent Governance

To optimize NetSuite application and database servers, the system employs certain mechanisms to control the consumption of web services.

Concurrency for REST web services is governed in a way that each request counts towards the account limit. The account governance limit applies to the combined total of web services and RESTlet requests. For detailed information about concurrency governance, see the help topic Web Services and RESTlet Concurrency Governance.

Additionally, if a request takes more than 15 minutes to complete, it automatically times out.

These mechanisms ensure the following:

- Requests are monitored and controlled to ensure that the user experience is not excessively impacted.
- The burden of heavy web services users is not shared among all users.
Working with REST Web Services Using Postman

In this document, the REST web services functionality is demonstrated using the Postman Application. However, you can use any similar tool of your preference to work with REST web services. Besides being able to build and send the API requests, the Postman Application also acts as a library for your requests which you can then import and export. You can download Postman at https://www.getpostman.com/

Installing Postman

Follow these steps to install Postman:

1. Download and install the Postman desktop application from https://www.getpostman.com/.

   **Note:** Do not use the (deprecated) Chrome extension, as it does not support some of the features that are present in the REST web services sample request collection.

2. Run Postman. Click Take me straight to the app on the initial splash screen.
3. Close the initial task window.

Working with Postman Environments and Collections

A Postman environment is a set of key-value pairs. The key represents the name of the variable. Using a Postman environment, you can switch between various NetSuite accounts, and between your test or production accounts. Using Postman environments, you can customize requests using variables so you can easily switch between different setups without changing your requests. You can also download environments, save them as JSON files, and upload them later.

A Postman collection is a set of HTTP requests. Similarly to the environments, you can create, share, duplicate, export, and delete a collection. You can also import a collection as a single JSON file. The collection distributed together with this document requires a proper environment setup as described in Importing and Setting Up a Postman Environment.

You can download the REST web services Postman environment template and collection of sample requests from the SuiteTalk tools download page at https://<accountID>.app.netsuite.com/app/external/integration/integrationDownloadPage.nl. To access the page, you must substitute your account ID in the URL.

   **Note:** To access the Postman environment template and collection, the REST Web Services feature must be enabled, and you must have the REST web services permission assigned to your role. For more information, see REST Web Services Prerequisites and Setup.

This set of sample requests for the Postman Application is provided to demonstrate how to use NetSuite's REST web services. The sample requests can also help you start building your REST-based integration with NetSuite.

For more information, see the following topics:
- Importing and Setting Up a Postman Environment
Importing and Setting Up a Postman Environment

To import a Postman collection:

1. Download the Postman environment template and collections archive from the SuiteTalk tools download page at https://<accountID>.app.netsuite.com/app/external/integration/integrationDownloadPage.nl. To access the page, you must substitute your account ID in the URL. To access the page, your role must have the REST web services permission assigned to it.

2. Unzip the archive.

3. To import the environment template from the Environment folder, click the gear icon in the top menu of the Postman application.

4. A popup window opens. Click Import on this window, and then select the template file to import.

5. After importing the environment, the REST WS Environment Template is displayed in the list of environments. Make a copy of the template for each new TBA token you want to add. Click the Duplicate button to make a copy.

6. Enter a self-descriptive name for your environment, for example, "<role> for <account> at <host>". REST web services only support account-specific domains. For information about account-specific domains, see the help topic URLs for Account-Specific Domains.

7. Enter your account ID (for example, 3604360). The host and protocol is filled in automatically.
8. Enter the TBA credentials you created. For details about creating TBA credentials, see Authentication.

---

**Managing Environments**

<table>
<thead>
<tr>
<th>Environment Name</th>
<th>NetSuite REST API Environment Template</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Variable** | **Initial Value** | **Current Value** | **Persist All** | **Reset All** |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>account</td>
<td>&lt;account&gt;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>host</td>
<td>&lt;account&gt;.suitetalk.net</td>
<td>&lt;account&gt;.suitetalkapi.netsuite.com</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proto</td>
<td>https</td>
<td>https</td>
<td></td>
<td></td>
</tr>
<tr>
<td>consumerKey</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>consumerSecret</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tokenId</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tokenSecret</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Importing the Postman Collection**

To import the Postman collection:

1. Download the REST web services Postman collection of sample requests from the SuiteTalk tools download page at https://<accountID>.app.netsuite.com/app/external/integration/integrationDownloadPage.nl. To access this page, you must substitute your account ID, and the REST web services permission must be assigned to your role.
2. Click Import in the top menu of the Postman Application, and select the previously downloaded collection (with *.json file extension).
3. Click the Collections tab in the left panel of the Postman Application to see the newly imported request collection.

**Sending a Request From the Imported Collection**

Send a test request to verify that your Postman environment configuration and collection import is correct.

To send a test request:

1. Click the label (NetSuite REST API Tutorial) of the newly imported collection. A list of requests is displayed.
2. Open the 0 Test folder in the imported collection, and select Example 0: Test Request from the list.
3. Click Send to execute the test request. The response to the request is displayed in the Response section of the Postman window.
Working with Resource Metadata

The REST API consists of a dynamic schema that is described by the metadata catalog. The metadata catalog serves as the API schema defining the contract, similarly to WSDL and XSD files in SOAP web services.

Using the metadata catalog, you can dynamically discover the API, including all available resources, the format and values of input and output, the supported HTTP methods and HTTP query parameters.

In NetSuite, records are the most important resource. The metadata catalog defines all available records, their fields, available values (for example, for enum fields), sublists and subrecords (both standard and custom ones), and their various properties. As new customizations are added to the system, they appear in the metadata catalog as well.

The metadata catalog for the beta version describes the following properties of the REST API:

- Names of records, fields, sublists, and subrecords
- Field types
- Reference fields and the referenced record types
- Enum fields and their available internal values
- Searchability of records, fields, sublists, and subrecords

The REST API comes with a fully personalized view on the resources (per user). This includes the ability to transparently work with user-specific NetSuite record customizations, such as custom records and fields. The ability to provide a record in its customized form means that the record structure can vary based on your specific NetSuite setup. Therefore, the REST API provides an option to dynamically generate metadata about the records (available resources and operations) in the form of standardized descriptions. Custom record metadata is accessible the same way as metadata for standard records.

Using metadata information, you can:

- Get an overview of all available record types
- Get an overview of the structure of a particular record type
- Get an overview of searchable fields
- Get an overview of supported HTTP methods and query parameters
- Automatically generate client code, for example, API client libraries or client stubs.

The beta version can provide metadata in Swagger (OpenAPI 3.0) and JSON Schema JSON-based formats. Both formats are used to define the structure of JSON data for validation, documentation, and interaction control.

The main difference between the two formats is that the metadata provided in JSON Schema format only describes the internal structure of a resource, for example, its fields, sublists, or subrecords.

In addition to this, the metadata in OpenAPI 3.0 format also describes links to related resources. It also describes how to interact with a resource through REST web services: it describes the URLs, HTTP methods, and parameters.

For more information, see the following help topics:

- Getting Metadata
- Working with OpenAPI 3.0 Metadata
Getting Metadata

The endpoint for getting the metadata schema for all exposed records is https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record where platform is the name of the REST web services API, v1 is the API version, and metadata-catalog/record is the resource that we are trying to access, that is, the record metadata.

Note: For detailed information about the URL schema for REST web services, see REST Web Services URL Schema and Account-Specific URLs.

The following is an example of a request that returns metadata about all records.

```
> GET https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record
```

The following is an example of an intentionally shortened response for the previous request:

```json
{
   "items": [
      {
         "name": "account",
         "links": [
            {
               "rel": "canonical",
               "href": "https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/account",
               "mediaType": "application/json"
            },
            {
               "rel": "alternate",
               "href": "https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/account",
               "mediaType": "application/swagger+json"
            },
            {
               "rel": "alternate",
               "href": "https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/account",
               "mediaType": "application/schema+json"
            },
            {
               "rel": "describes",
               "href": "https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/record/account"
            }
         ]
      }
   ]
}
```

The response informs you through HATEOAS links about the possible mediaType flavor in which the response can be obtained. You can see that the metadata for each record can be served in both OpenAPI 3.0 and JSON schema formats.

SuiteTalk REST Web Services (Beta)
Working with OpenAPI 3.0 Metadata

To get the metadata in OpenAPI 3.0 format, you have to specify the proper value in the Accept HTTP header. For OpenAPI 3.0, you must add a header in the format `Accept: application/swagger+json`.

To avoid the long loading time needed to gather OpenAPI 3.0 metadata for all records, the metadata catalog resource supports the `select` query parameter for selecting particular record types. You can use this parameter to restrict the metadata to certain record types only. In the following example, the request is modified to restrict the metadata to customer and sales order records only.

```
> SET-HEADER Accept: application/swagger+json
> GET https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record?
select=customer,salesorder
```

The following is an intentionally shortened response to the previous request.

```
"paths": {
  "/salesorder": {
    "get": {
      "tags": ["salesorder"],
      "summary": "Get list of records",
      "parameters": [
        {
          "name": "q",
          "in": "query",
          "description": "Search query used to filter results",
          "schema": {
            "type": "string"
          }
        }
      ],
      "responses": {
        "200": {
          "description": "Get record",
          "content": {
            "application/vnd.oracle.resource+json; type=collection": {
              "schema": {
                "$ref": "#/components/schemas/salesorderCollection"
              }
            }
          }
        },
        "default": {
          "description": "Error response.",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/rest_error"
              }
            }
          }
        }
      }
    }
  }
}
```
The response describes the structure of the records as well as the supported operations, HTTP methods, and query parameters.

The following image shows an excerpt from the response. It describes the record name, the query parameters, and the available HTTP methods you can use with the record.

```
"paths": {
  "/customers": { },
  "/salesorder": { 
    "get": { 
      "tags": [ "salesorder" ],
      "summary": "Get list of records",
      "parameters": [ 
        { 
          "name": "q",
          "in": "query",
          "description": "Search query used to filter results",
          "required": false,
          "schema": { 
            "type": "string"
          }
        }
      ],
      "responses": {
        "200": { },
        "400": { }
      }
    },
    "/customer/(id)": { },
    "/salesorder/(id)": { }
  }
}
```

The following excerpt describes the record structure and the properties of different fields: their name, type, and attributes, and whether they are searchable.

```
"salesorder": { 
  "type": "object",
  "properties": { 
    "defaultIlShipMethKey": { 
      "title": "[Missing label:defaultIlShipMethKey]",
      "type": "number",
      "format": "integer",
      "x-mx-searchable": false
    },
    "intercossuppressereventsandemails": { 
      "title": "[Missing label:intercossuppressereventsandemails]",
      "type": "string",
      "x-mx-searchable": false
    },
    "startDate": { 
      "title": "Start Date",
      "type": "string",
      "format": "date",
      "x-mx-searchable": false
    }
  }
```

The following excerpt describes a sublist of the sales order record.
The OpenAPI 3.0 record metadata received in the previous step can be used as input for any OpenAPI 3.0 compatible tool for further processing. For example, you can use the metadata in tools that generate REST client stubs as well as tools that specialize in dumping a static webpage that describes record structure. In this example, OpenAPI 3.0 metadata is used in the freely available online Swagger Editor for exploring the structure and operations that can be done with the customer and sales order records.

**Note:** The performance of the online Swagger editor can be limited if there is a large amount of data to process. Therefore it is not recommended to use the editor to generate metadata description for all exposed records.

By copy-pasting the example response to the Swagger Editor, you can see an output similar to the following:

```
"salesorder-giftCertRedemptionElement": { 
  "type": "object",  
  "properties": {  
    "authCodeApplied": {  
      "title": "Amount Applied",  
      "type": "number",  
      "format": "double",  
      "x-ns-searchable": true  
    },  
    "authCodeRemaining": {  
      "title": "Available Credit",  
      "type": "number",  
      "format": "double",  
      "x-ns-searchable": false  
    },  
    "parentTransaction": {  
      "$ref": "#/components/schemas/salesorder"  
    },  
    "authCode": {  
      "$ref": "#/components/schemas/x-ns-generic"  
    }  
  } 
},
```

The OpenAPI 3.0 record metadata received in the previous step can be used as input for any OpenAPI 3.0 compatible tool for further processing. For example, you can use the metadata in tools that generate REST client stubs as well as tools that specialize in dumping a static webpage that describes record structure. In this example, OpenAPI 3.0 metadata is used in the freely available online Swagger Editor for exploring the structure and operations that can be done with the customer and sales order records.

**Note:** The performance of the online Swagger editor can be limited if there is a large amount of data to process. Therefore it is not recommended to use the editor to generate metadata description for all exposed records.

By copy-pasting the example response to the Swagger Editor, you can see an output similar to the following:

```
Note: Links to NetSuite records that are out of the scope of the select parameter are declared as generic JSON objects with no structure in the returned record metadata. You can avoid this by listing the referenced record in the select parameter.
```

From the graphic representation, you can see how to obtain a list of all customer and sales order records, how to create new record instances, and how to perform read, update, and delete operations upon them. You can expand each REST method for more description. The Swagger Editor output also contains a **Schema** section. The figure below shows how to use it to explore the structure of a record. For instance, you can see that the salesorder record contains the item sublist, represented in the form of salesorder-itemCollection that contains the **totalResults**, links, and items properties. The structure of a single line of the item sublist is then captured in the salesorder-itemElement part.
Working with OpenAPI 3.0 Metadata

Similarly to OpenAPI 3.0 metadata, you can obtain the JSON Schema description by setting the value of the Accept HTTP header to `application: schema+json`, as in the following example.

```bash
> SET-HEADER Accept: application/schema+json
> GET https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/customer
```

The following is a shortened example of a response to the previous request. Note that the metadata provided in JSON Schema format only describes the internal structure of a resource, for example, its fields, sublists, or subrecords, and links to related resources, and does not describe URLs, HTTP methods, and parameters.

```json
{
  "type": "object",
  "properties": {
    "startdate": {
      "title": "Start Date",
      "type": "string",
      "format": "date",
      "x-netsuite-searchable": true
    },
    "entityid": {
      "title": "Entity ID",
      "type": "string",
      "x-netsuite-searchable": true
    },
    "parent": {
      "type": "object",
```
"properties": {
  "id": {
    "title": "id",
    "type": "string",
  },
  "rest_name": {
    "title": "rest_name",
    "type": "string",
    "readOnly": true
  }
}

"$schema": "https://json-schema.org/draft-06/hyper-schema#"

Note: The JSON generated by the beta version of the REST API generates external URL links leading to metadata about the linked record types. These are, however, reachable only locally and are unreachable for external validators (such as the JSON Schema validator).

The JSON Schema record metadata received in the previous step can be used as input for any JSON Schema Draft 6 compatible library and tool for further processing. The most frequent use of the JSON Schema record metadata is input and output validation. The following JSON object is an example representing a customer record.

```json
{
  "entityid": "My Customer",
  "currency": { "id": "1" },
  "representingsubsidiary": { "id": "1" },
  "monthlyclosing": "31 - End of the Month"
}
```

You can check if this record representation conforms to the record metadata schema by using the freely available JSON Schema Validator.
No errors found. JSON validates against the schema
Working with Records

Using REST web services, you can perform CRUD (create, read, update, delete) operations on NetSuite records. The following sections provide information about the structure of NetSuite records, and the ways you can work with records using REST web services.

- NetSuite Record Structure
- Creating a Record Instance
- Getting a Record Instance
- Updating a Record Instance
- Using the Upsert Operation
- Deleting a Record Instance
- Using External IDs

NetSuite Record Structure

The figure below shows a standard customer record in NetSuite.

The figure outlines the basic components of the record:

1. Body fields - for example, name, ID, or email. Body fields are placed either on the main area of the record or on a subtab.
2. Sublists - for example, the Info sublist on a customer record. A sublist consists of line items and their fields. For more information, see the help topic What is a Sublist?
3. Subrecords - for example, the address subrecord. A subrecord includes many of the same elements as standard NetSuite records, for example, body fields, sublists, and sublist fields. However, you can only create, edit, remove, or view a subrecord from within the context of its parent record. For more information, see the help topic What is a Subrecord?

You can use REST web services to get and set values on body fields and sublists. You can also use REST web services to create these components.

Creating, updating, and deleting subrecords is not supported in the beta version.
Creating a Record Instance

In REST web services, you can create a new record using the POST HTTP method. The POST method expects a request body (a JSON object) that conforms to the record's metadata schema and contains values for at least each mandatory field of the given record type.

The REST beta version does not support the update of subrecords.

**Note:** Not all record fields can be set using the POST method. For example, the id field is read-only.

Any omitted fields are considered empty or to have default values. The following example shows how to create a new customer record instance.

```bash
> POST https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer
> { "entityid": "New Customer", "companyname": "My Company", "subsidiary": { "id": "1" } }
```

The following is an excerpt from the response headers of a successful operation, with no body content returned (HTTP Code 204). The URL of the newly created record is given in the Location header of the response.

```
Date #Fri, 04 Jan 2019 08:50:20 GMT
Location #https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/647
Content-Type #application/json
```

Getting a Record Instance

In the following example, an instance of the customer record type created in the previous section is retrieved. You can get a particular customer instance by sending a request in the following form: /rest/platform/v1/record/customer/<id>. In the following example, the id of the newly created record is 107. You can read the record instance using the following request.

```bash
> GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107
```

The following is an excerpt from the response headers of a successful operation, returned with HTTP Code 200.

```
Date #Thu, 14 Feb 2019 14:05:27 GMT
Content-Length #6246
Content-Type #application/vnd.oracle.resource+json; type=singular
```

The following is a shortened example of the body of the response.

```json
{
  "links": [
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107"
    }
  ],
  "referrerlist": {
```

SuiteTalk REST Web Services (Beta)
The response is in a format that conforms to the Swagger and JSON schema. You can validate the retrieved data against the JSON Schema. For information about validation, see Working with JSON Schema Metadata.

In the response, note the Hypermedia As The Engine Of Application State (HATEOAS) links elements. Using the links elements, you can navigate through the REST endpoint.

Format of Sublists and Subrecords

The REST web services beta version automatically expands all sublists and subrecords. For instance, note the expansion of the addressbook sublist and the addressbookaddress subrecord on the customer record.
Getting a Record Instance

```json
{
  "items": [
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107/addressbook/1"
        }
      ],
      "addressbookaddress_text": "Alan Rath
Glenrock General Hospital
417 Washington Blvd
Glenrock WY 82637",
      "label": "Bill to",
      "addressid": "39",
      "addressbookaddress_type": "addr",
      "internalid": 39,
      "isnewline": false,
      "defaultbilling": false,
      "defaultshipping": false,
      "isresidential": false,
      "id": "39",
      "addressbookaddress_key": "152"
    }
  ],
  "totalResults": "1"
}
```

Format of Selects and References

References to records in the REST web services beta version contain an internal ID (id), a reference name (refName) (that is, the value obtained by invoking the getFieldText() method on the reference field), and an HATEOAS link navigating to the referenced record.

```json
"currency": {
  "links": [
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/currency/1"
    }
  ],
  "refName": "USD",
  "id": "1"
}
```

Format of Multiselects

The REST web services beta version models multiselects as a collection of references containing the internal ID and HATEOAS links navigating to the referenced record. The following example shows the subsidiary multiselect field that is located on the account record.

```json
"subsidiary": {
  "links": [
```
Getting a Record Instance

```
{
  "rel": "self",
  "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/account/106/subsidiary"
}
],
"items": [
{
  "links": [
  
  ],
  "id": "1"
},
{
  "links": [
  
  ],
  "id": "2"
},
"totalResults": 2
}
```

Format of Enumeration Values

The beta version of REST web services uses **internal** values of enumeration values. This means the values that are passed to and from SuiteScript by the get-/setFieldValue() functions.

Updating a Record Instance

In the following example, the name of a specific customer record instance is updated. In REST web services, you can perform such an update using the PATCH HTTP method. The PATCH method expects a request body with the same fields that can be retrieved using the GET method. That is, the record's metadata schema is shared between reads and updates.

**Note:** Not all record fields can be updated using the PATCH method. For example, the id field is considered to be read-only.

Any omitted fields are considered unchanged.

In this example, the name (entityid) of the record instance 107 retrieved in the previous example is changed from "Alan Rath" to "Updated Customer". Send the following request to perform the update.

```shell
> PATCH https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107
> BODY { "entityid": "Updated Customer" }
```
The following is an excerpt from the response headers of a successful operation, with no body content returned (HTTP Code 204). The URL of the updated record is given in the Location header of the response.

```
Date #Fri, 04 Jan 2019 09:07:01 GMT
Location #https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107
Content-Type #application/json
```

### Using the Upsert Operation

The upsert operation enables you to either add a record that does not exist, or update an existing record. You can only use the upsert operation when you use an external ID in the request URL and when you use the PUT HTTP method. For information about using external IDs, see Using External IDs.

You can use the upsert operation as a synchronization tool. When using the upsert operation, you do not need to be concerned whether the record with the given external ID already exists.

The following example shows how to use the upsert operation. If the record does not exist, it will be added. If the record already exists, it will be updated.

```
> PUT http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/eid:C1D002
> {  
>   "firstName": "John",
>   "lastName": "Smith"
> }
```

### Deleting a Record Instance

To delete a record instance, you need to specify the record type and the instance identifier. The following is a delete request.

```
> DELETE https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107
```

The following is an excerpt from the response headers of a successful operation, with no body content returned (HTTP Code 204).

```
Date #Thu, 14 Feb 2019 15:38:21 GMT
Content-Type #application/json
```

### Using External IDs

Each record in NetSuite can be uniquely identified by its record type in combination with either an external ID or a system-generated internal ID. For an overview of internal and external IDs, see the help topic Using Internal IDs, External IDs, and References.

You can use an external ID as a key to a record instead of an internal ID. The main use of external IDs is during synchronization with existing data outside of NetSuite.

In REST web services, an external ID starts with the prefix "eid:" in the following format:

`eid:external_id`
An external ID can be any string containing letters, numbers, underscore (_), and hyphen (-).
You can use external IDs anywhere in the URL where an internal ID can be used. You can also use an external ID in the request body with the field name `externalId`.

The following example adds a record. The external ID is used in the request body.

```
> POST http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer
> BODY {
>   "firstName": "John",
>   "lastName": "Smith",
>   "isPerson": true,
>   "externalId": "CID001",
>   "subsidiary": {
>     "id": "1"
>   }
> }
```

The following example retrieves a record. External IDs are used in the request to identify the record.

```
GET http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/eid:CID001/subscriptions/
  eid:SUID042
```
Working with Sublists

Sublists are an important part of several NetSuite record types. In the REST web services beta, you can use the POST and PATCH operations to access a sublist on a record instance. However, because sublists in NetSuite are of multiple types, for instance, editable sublist, applicable sublist, or search sublist, each sublist can have slightly different behavior. Some sublists on particular records can even contain pre-generated lines. Moreover, if a line is added based on user input, it can contain default or computed values. Some sublists include one or more key fields that can uniquely identify each sublist record. These are keyed sublists. The behavior of keyed sublists is different from the behavior of non-keyed sublists.

The following list includes all operations that can be performed with a sublist:

1. **Add new**
   - Single line
   - Multiple lines at once
2. **Update existing**
   - Single line
   - Unset particular field
   - Multiple lines at once
3. **Remove existing**
   - Single line
   - Multiple lines at once
   - All
4. **Mix (any combination of the above at once)**
5. **Replace all**

Operations 2.a, 2.b, 2.c, 3.a, 3.b, and 4 are not possible on non-keyed sublists. You can achieve the equivalent of these operations by replacing all sublist lines.

SOAP web services offer a special `replaceAll` attribute to enable some of these operations. For more information about this topic, see the help topic *Updating Sublists in SOAP Web Services*.

Similar behavior is also reflected in REST web services. This section provides details about each sublist operation mode.

In all the examples in this section, assume the following initial state of a record instance (unless otherwise specified):

```json
GET .../myrecord/100

{
   "body1": "previous body text 1",
   "body2": "previous body text 2",
   "sublist": {
      "items": [
         { "key1": "a", "key2": "1", "col": "previously present line 1" },
         { "key1": "b", "key2": "2", "col": "previously present line 2" },
         { "key1": "X", "key2": "0", "col": "previously present line 0" }
      ]
   },
   "unkeyedsublist": {
      "items": [
      ]
   }
}
```
Creating a Sublist

During create operations, lines of a keyed sublist are updated if the line contains matching keys, and the remaining lines from the request are added. For non-keyed sublists, the lines are always appended to the ones that are already present on the sublist by default. The following is an example of a POST operation upon a sublist.

Request Body

POST .../myrecord

```json
{
  "body1": "inserted body text 1",
  "sublist": {
    "items": [
      { "key1": "a", "key2": "1", "col": "inserted line 1" },
      { "key1": "b", "key2": "2", "col": "inserted line 2" }
    ]
  },
  "unkeyedsublist": {
    "items": [
      { "col": "inserted line 1" },
      { "col": "inserted line 2" }
    ]
  }
}
```

Response – Post State

Response:

201 Created

Location: .../myrecord/101

```json
{
  "body1": "inserted body text 1",
  "body2": "default body text 2",
  "sublist": {
  
  }
}
```
Creating a Sublist

```json
"items" : [
    { "key1": "a", "key2": "1", "col": "inserted line 1" },
    { "key1": "b", "key2": "2", "col": "inserted line 2" },
    { "key1": "X", "key2": "0", "col": "default line 0" },
],
"unkeyedsublist" : {
    "items" : [
        { "col": "default line 1" },
        { "col": "default line 2" },
        { "col": "default line 0" },
        { "col": "inserted line 1" },
        { "col": "inserted line 2" },
    ],
},
}
```

**Note:** Keyed sublist lines are updated if the key matches, otherwise, a new line is created (in other words, sublist lines are merged). Lines for non-keyed sublists are added.

Updating a Sublist

During update operations, lines of a keyed sublist are updated if the line contains matching keys, and the remaining lines from the request are added. For non-keyed sublists, the lines are always appended to the ones that are already present on the sublist. The following examples show PATCH operations upon a sublist.

Request Body – Patch Operation

PATCH .../myrecord/100

```json
{
    "body1": "replaced body text 1",
    "sublist" : {
        "items" : [
            { "key1": "a", "key2": "1", "col": "replaced line 1" },
            { "key1": "b", "key2": "2", "col": "replaced line 2" }
        ],
    },
    "unkeyedsublist" : {
        "items" : [
            { "col": "inserted line 1" },
            { "col": "inserted line 2" },
        ]
    }
}
```

Response – Post State

Response:
Updating a Sublist

204 No Content

Post State:

```
{
    "body1": "replaced body text 1",
    "body2": "previous body text 2",
    "sublist" : {
        "items" : [
            { "key1": "a", "key2": "1", "col": "replaced line 1" },
            { "key1": "b", "key2": "2", "col": "replaced line 2" },
            { "key1": "X", "key2": "0", "col": "previously present line 0" },
        ]
    },
    "unkeyedsublist": {
        "items" : [
            { "col": "previously present line 1" },
            { "col": "previously present line 2" },
            { "col": "previously present line 0" },
            { "col": "inserted line 1" },
            { "col": "inserted line 2" },
        ]
    }
}
```

**Note:** Keyed sublist lines are updated if the key matches, otherwise, a new line is created (in other words, sublist lines are merged). All lines of non-keyed sublists are added.

Request Body – Nullifying a Sublist

PATCH .../myrecord/100

```
{
    "body1": "replaced body text 1",
    "sublist": {
        "items": null
    }
}
```

```
{
    "body1": "replaced body text 1",
    "sublist": null
}
```

Response – Post State

Response:

204 No Content

Post State:

```
{
}
```
Updating a Sublist

29

"body1": "replaced body text 1",
"body2": "previous body text 2",
"sublist" : {
   "items" : []
},
"unkeyedsublist" : {
   "items" : [
      { "col": "previously present line 1" },
      { "col": "previously present line 2" },
      { "col": "previously present line 0" }
   ]
}

**Note:** Keyed as well as non-keyed sublist lines are replaced. If the sublist is mandatory, it is expected that the operation fails.

Replacing a Sublist

During sublist replacements, lines of a keyed sublist are updated if the line contains matching keys, other lines are removed, and the remaining unmatched lines from the request are added. For non-keyed sublists, all lines are always removed and replaced by lines that are in the incoming request. The following examples show POST and PATCH operations with a replace query parameter.

Request Body – Replacing a Default Keyed Sublist

POST .../myrecord?replace=sublist

```
{
   "body1": "inserted body text 1",
   "sublist" : {
      "items" : [
         { "key1": "a", "key2": "1", "col": "inserted line 1" },
         { "key1": "b", "key2": "2", "col": "inserted line 2" }
      ],
   },
   "unkeyedsublist" : {
      "items" : [
         { "col": "inserted line 1" },
         { "col": "inserted line 2" }
      ]
   }
}
```

Response – Post State

Response:

201 Created

Location: .../myrecord/101
Replacing a Sublist

```
{
  "body1": "inserted body text 1",
  "body2": "default body text 2",
  "sublist": {
    "items": [
      { "key1": "a", "key2": "1", "col": "inserted line 1" },
      { "key1": "b", "key2": "2", "col": "inserted line 2" },
    ]
  },
  "unkeyedsublist": {
    "items": [
      { "col": "default line 1" },
      { "col": "default line 2" },
      { "col": "default line 0" },
      { "col": "inserted line 1" },
      { "col": "inserted line 2" },
    ]
  }
}
```

**Note:** Keyed sublist lines are updated if the key matches, other lines are removed, and unmatched lines from the request are created as new. All lines of non-keyed sublists are added. The operation fails if any replaced default sublist line is read-only.

Request Body – Replacing a Default Non-Keyed Sublist

POST .../myrecord?replace=unkeyedsublist

```
{
  "body1": "inserted body text 1",
  "sublist": {
    "items": [
      { "key1": "a", "key2": "1", "col": "inserted line 1" },
      { "key1": "b", "key2": "2", "col": "inserted line 2" }
    ]
  },
  "unkeyedsublist": {
    "items": [
      { "col": "inserted line 1" },
      { "col": "inserted line 2" }
    ]
  }
}
```

Response – Post State

Response:

201 Created

Location: .../myrecord/101
Replacing a Sublist

```
{
    "body1": "inserted body text 1",
    "body2": "default body text 2",
    "sublist": {
        "items": [
            { "key1": "a", "key2": "1", "col": "inserted line 1" },
            { "key1": "b", "key2": "2", "col": "inserted line 2" },
            { "key1": "X", "key2": "0", "col": "default line 0" }
        ],
    },
    "unkeyedsublist": {
        "items": [
            { "col": "inserted line 1" },
            { "col": "inserted line 2" },
        ]
    }
}
```

**Note**: Keyed sublist lines are updated if the key matches, otherwise, a new line is created. All lines of non-keyed sublists are removed and then created as new. The operation fails if any replaced default sublist line is read-only.

Request Body – Replacing a Keyed Sublist

```
PATCH ../myrecord/100?replace=sublist
```

```
{
    "body1": "replaced body text 1",
    "sublist": {
        "items": [
            { "key1": "a", "key2": "1", "col": "replaced line 1" },
            { "key1": "b", "key2": "2", "col": "replaced line 2" }
        ],
    },
    "unkeyedsublist": {
        "items": [
            { "col": "inserted line 1" },
            { "col": "inserted line 2" },
        ]
    }
}
```

Response – Post State

Response:

```
204 No Content
```

Post State:

```
{
    "body1": "replaced body text 1",
```
Replacing a Sublist

```
"body2": "previous body text 2",
"sublist": {
  "items": [
    { "key1": "a", "key2": "1", "col": "replaced line 1" },
    { "key1": "b", "key2": "2", "col": "replaced line 2" },
  ],
},
"unkeyedsublist": {
  "items": [
    { "col": "previously present line 1" },
    { "col": "previously present line 2" },
    { "col": "previously present line 0" },
    { "col": "inserted line 1" },
    { "col": "inserted line 2" },
  ]
}
```

**Note:** Keyed sublist lines are updated if the key matches, other lines are removed, and unmatched lines from the request are created as new. All lines of non-keyed sublists are added.

Request Body – Replacing a Non-Keyed Sublist

PATCH ..:/myrecord/100?replace=unkeyedsublist

```
{
  "body1": "replaced body text 1",
  "sublist": {
    "items": [
      { "key1": "a", "key2": "1", "col": "replaced line 1" },
      { "key1": "b", "key2": "2", "col": "replaced line 2" }
    ],
  },
  "unkeyedsublist": {
    "items": [
      { "col": "inserted line 1" },
      { "col": "inserted line 2" }
    ]
  }
}
```

Response – Post State

Response:

204 No Content

Post State:

```
{
  "body1": "replaced body text 1",
  "body2": "previous body text 2",
}
Replacing a Sublist

```
"sublist" : {
    "items" : [
        { "key1": "a", "key2": "1", "col": "replaced line 1" },
        { "key1": "b", "key2": "2", "col": "replaced line 2" },
        { "key1": "X", "key2": "0", "col": "previously present line 0" },
    ],
},
"unkeyedsublist" : {
    "items" : [
        { "col": "inserted line 1" },
        { "col": "inserted line 2" },
    ]
}
```

**Note:** Keyed sublist lines are updated if the key matches, otherwise, a new line is created (in other words, sublist lines are merged). All lines of non-keyed sublists are removed and then created as new.

Request Body – Replacing Multiple Sublists

The following operation is a combination of the previous operations.

PATCH .../myrecord/100?replace=sublist,unkeyedsublist

```
{
    "body1": "replaced body text 1",
    "sublist" : {
        "items" : [
            { "key1": "a", "key2": "1", "col": "replaced line 1" },
            { "key1": "b", "key2": "2", "col": "replaced line 2" }
        ],
    },
    "unkeyedsublist" : {
        "items" : [
            { "col": "inserted line 1" },
            { "col": "inserted line 2" },
        ]
    }
}
```

Response – Post State

Response:
204 No Content

Post State:

```
{
    "body1": "replaced body text 1",
    "body2": "previous body text 2",
    "sublist" : {
```

SuiteTalk REST Web Services (Beta)
Replacing a Sublist

```json
"items" : [ 
  { "key1": "a", "key2": "1", "col": "replaced line 1" },
  { "key1": "b", "key2": "2", "col": "replaced line 2" },
],
"unkeyedsublist" : { 
  "items" : [ 
    { "col": "inserted line 1" },
    { "col": "inserted line 2" },
  ]
}
}
Working with Subrecords

Subrecords represent a way of storing data in NetSuite. Like records, subrecords are classified by type. Some common types of subrecord include address, inventory detail, and order schedule.

A subrecord includes many of the same elements of a standard NetSuite record (body fields, sublists and sublist fields, and so on). However, subrecords must be created, edited, removed, or viewed from within the context of a standard (parent) record.

The purpose of a subrecord is to hold key related data about the parent record. For example, a parent record would be a Serialized Inventory Item record. This record defines a type of item. A subrecord would be an Inventory Detail subrecord. This is a subrecord that contains all data related to where the item might be stored in a warehouse. In this way, the subrecord contains data related to the item, but not data that directly defines the item. Without the parent record, the subrecord would serve no purpose. For more information about subrecords in general, see the help topics What is a Subrecord? and Understanding Subrecords.

In REST web services, you can use the POST and PATCH operations to access a subrecord on the record instance.

Updating a Subrecord

Subrecords are modelled as inner JSON properties in the request and response body. In the following example, a subrecord, `addressbookaddress` is set on the sublist line of the `addressbook` sublist.

```
> POST http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer

{  
  "entityid": "My Customer",
  "companyname": "My Company",
  "email": "another.customer@company.com",
  "addressbook": {
    "items": [
      {
        "label": "New York HQ",
        "addressbookaddress": {
          "country": "US",
          "state": "NY",
          "zip": "10001",
          "addressee": "Dwight Schrute"
        }
      }
    ]
  }
}
```

Getting a Subrecord

The following example shows a GET operation, which is used to access a subrecord.

```
> GET http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/42/addressbook/24/addressbookaddress

{
  "addressee": "Dwight Schrute",
}
```
The structure of the subrecord is also returned on the parent record, on the sublist line, if you set the `expandSubResources` query parameter to `true`. 
Record Filtering and Query

The query operation is used to execute a query on a specific record type based on a set of criteria. Record query only returns record IDs and HATEOAS links. That is, query results have a form of non-expanded references. Additionally, you can only use body fields in query conditions. Saved queries, multilevel joins, and sublist and subrecord queries are not supported in the beta version.

The beta version of REST web services only supports limited record query. Joins are not supported. For more information, see the following topics:

- Listing All Record Instances
- Record Collection Filtering
- Collection Paging

Listing All Record Instances

You can obtain the list of all records of a record type by sending an HTTP GET request to .../rest/platform/v1/record/<record_type>, as shown in the following image.

Listing all instances of a given record type can also be understood as a query over the given record type without any criteria. The following is an example of such a request.

> GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer

The following is a shortened example of a response.
Listing All Record Instances

```
{
  "links": [
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer"
    }
  ],
  "items": []
}.
```

Record Collection Filtering

You can filter the collection of all record instances by using the q query parameter to specify filter conditions. Each condition consists of a field name, a filter, and constraint. You can join several conditions using AND / OR operators. The following table contains a shortened list of available query filters with their associated field types.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Allowed Filters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>IS</td>
</tr>
<tr>
<td>Double, Integer, Float, Number, Duration</td>
<td>EQUAL, EQUAL_NOT, GREATER, GREATER_NOT, GREATER_OR_EQUAL, GREATER_OR_EQUAL_NOT, LESS, LESS_NOT, LESS_OR_EQUAL, LESS_OR_EQUAL_NOT</td>
</tr>
</tbody>
</table>

SuiteTalk REST Web Services (Beta)
The following is an example of a simple query.

**Note:** The spaces in URLs are encoded. The following examples are presented without encoding for clarity.

```
GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?q=email START_WITH barbara
```

The response is a collection of customer record instances where the value in the email field starts with the value of barbara. The result is a collection resource containing links to resources that match query criteria. The response could be similar to the following:

```
{
    "links": [
        {
            "rel": "self",
            "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer"
        }
    ],
    "items": [
        {
            "links": [
                {
                    "rel": "self",
                    "href": "https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107"
                }
            ],
            "id": "107"
        }
    ],
    "totalResults": 1
}
```

When your condition constraint contains spaces, you should use quotation marks around the constraint, for instance, firstname IS “Barbara Allen”. See the following additional query examples:

- Find customer by company name (string constraint):
  GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?q=companyname START_WITH "Another Company"

- Find inactive customers (boolean constraint):
  GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?q=isinactive IS true

- Find customers created in 2019 (date constraint, AND operator):
  GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?q=dateCreated ON_OR_AFTER "1/1/2019" AND dateCreated BEFORE "1/1/2020"

- Find customers with high or low credit limit (number constraint, OR operator):
Record Collection Filtering

GET https://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?q=creditlimit\nGREATER_OR_EQUAL 1000 OR creditlimit LESS_OR_EQUAL 10

Collection Paging

Lists of record instances and the results of saved analytics workbook searches are returned in one or more pages. The results are displayed on multiple pages, with the default setting of 1000 results per page. You can also specify your own paging by adding the limit criteria to your request.

The following is an example of a request for all instances of customer records, displaying two results per page:

> GET http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?limit=2

The following is an example of a response:

```
{
  "links": [
  {
    "rel": "next",
    "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/account?limit=2&offset=2"
  },
  {
    "rel": "last",
    "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/account?limit=22&offset=2"
  },
  {
    "rel": "self",
    "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?limit=2&offset=0"
  }
  ],
  "items": [
  {
    "links": [
    {
      "rel": "self",
      "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/107"
    }
    ],
    "id": "107"
  },
  {
    "links": [
    {
      "rel": "self",
      "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/41"
    }
    ],
    "id": "41"
  }
  ],
  "count": 2,
  "offset": 0,
  "hasMore": true,
}
The values of the `count`, `hasMore`, and `totalResults` fields provide information about the results that can be retrieved in subsequent requests for other pages. Links to the last and next pages provide direct links that you can use to retrieve these pages.

You can retrieve the next page by sending a request with the offset value specified. The offset provides the index of the result from which you should start the requested page.

**Note:** You can only set offset and limit values in a way that the offset is divisible by the limit. For example, Offset=20, Limit=10 or Offset=0, Limit=5

The following is an example of a request for the second (and last) page of results with all customers:

```
> GET http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?limit=2&offset=2
```

The following is an example of a response:

```
{
  "links": [
    {
      "rel": "prev",
      "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/account?limit=2&offset=0"
    },
    {
      "rel": "first",
      "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/account?limit=22&offset=0"
    },
    {
      "rel": "self",
      "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer?limit=2&offset=2"
    }
  ],
  "items": [
    {
      "links": [
        {
          "rel": "self",
          "href": "http://demo123.suitetalk.api.netsuite.com/rest/platform/v1/record/customer/90"
        }
      ],
      "id": "90"
    }
  ],
  "count": 1,
  "offset": 2,
  "hasMore": false,
  "totalResults": 3
}
```
Error Handling in REST Web Services

In REST web services, HTTP status codes are used to inform you about the success or failure of a request. The following status codes are used.

- 2xx status codes are used for successful requests.
- 4xx status codes are used for failures due to user error.
- 5xx status codes are used for failures due to system error.

The following examples show some common errors with their descriptions.

Missing Login Information

The following error is returned if the request does not contain the login authorization header.

```json
{
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
  "title": "Missing login header!",
  "status": 400,
  "o:errorCode": "USER_ERROR"
}
```

Request for Non-existent Record

The request GET https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/customer/abc returns the following error, because the request is for a non-existent customer.

```json
{
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
  "title": "Invalid record instance identifier (i.e., id, external id, or name id) abc in request URL",
  "status": 400,
  "o:errorCode": "USER_ERROR"
}
```

Invalid Request

The request GET https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/customer, salesorder returns the following error, because there is an extra space in the query parameters. The correct request would be https://DEMO123.suitetalk.api.netsuite.com/rest/platform/v1/metadata-catalog/record/customer,salesorder.

```json
{
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
  "title": "The request could not be understood by the server due to malformed syntax.",
  "status": 400,
  "o:errorCode": "INVALID_REQUEST"
}
```
Exceeded Concurrency Governance Limit

The following error is returned if the request is rejected due to exceeding the limit allowed by concurrency governance. For information about request limits, see Concurrency Governance.

```
{
    "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
    "title": "Concurrent request limit exceeded. Request blocked.",
    "status": 429,
    "o:errorCode": "USER_ERROR"
}
```

System Error

The following error is returned if a system error occurs while the request is being processed. If a system error occurs, contact NetSuite Customer Support and refer to the error ID.

```
{
    "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
    "title": "An unexpected error occurred. Error ID: jrgbpyylphishmmlxyt",
    "status": 500,
    "o:errorCode": "UNEXPECTED_ERROR"
}
```