SuiteTalk REST Web Services
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

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are “commercial computer software” or “commercial computer software documentation” pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any
operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government’s use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

This documentation is in pre-General Availability status and is intended for demonstration and preliminary use only. It may not be specific to the hardware on which you are using the software. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to this documentation and will not be responsible for any loss, costs, or damages incurred due to the use of this documentation.

The information contained in this document is for informational sharing purposes only and should be considered in your capacity as a customer advisory board member or pursuant to your pre-General Availability trial agreement only. It is not a commitment to deliver any material, code, or functionality, and 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.

**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 ................................................................................ 3  
REST Web Services Prerequisites and Setup .......................................................................................... 4  
REST Web Services URL Schema and Account-Specific URLs ............................................................... 6  
Authentication for REST Web Services ................................................................................................. 9  
Working with REST Web Services Using Postman .................................................................................. 11  
   Importing and Setting Up a Postman Environment ................................................................................. 12  
   Importing the Postman Collection ......................................................................................................... 13  
   Sending a Request From the Imported Collection ................................................................................. 14  
Working with Resource Metadata ........................................................................................................ 15  
   Getting Metadata ................................................................................................................................. 16  
   Working with OpenAPI 3.0 Metadata ..................................................................................................... 16  
   Working with JSON Schema Metadata ............................................................................................... 20  
Working with Records .......................................................................................................................... 23  
   The REST API Browser ......................................................................................................................... 23  
   NetSuite Record Structure .................................................................................................................... 25  
   Using CRUD Operations on Custom Records, Custom Lists, and Custom Transactions v1 .................... 26  
   Creating a Record Instance .................................................................................................................... 27  
   Getting a Record Instance ..................................................................................................................... 27  
      Format of Sublists and Subrecords ....................................................................................................... 28  
      Format of Selects and References ....................................................................................................... 29  
      Format of Multiselects ......................................................................................................................... 29  
      Format of Enumeration Values ........................................................................................................... 30  
   Updating a Record Instance ................................................................................................................... 30  
   Using the Upsert Operation .................................................................................................................... 31  
   Deleting a Record Instance .................................................................................................................... 31  
   Accessing Subresources in REST Web Services ..................................................................................... 31  
   Using External IDs ............................................................................................................................... 32  
   Using Datetime Fields ........................................................................................................................... 33  
   Executing Record Actions ..................................................................................................................... 34  
   Transforming Records .......................................................................................................................... 35  
   Working with the Pricing Sublist on Item Records ............................................................................... 36  
   Using the REST Web Services SuiteScript Execution Context ............................................................. 37  
Working with Sublists ........................................................................................................................... 38  
   Creating a Sublist ................................................................................................................................. 39  
   Updating a Sublist ............................................................................................................................... 40  
   Replacing a Sublist ............................................................................................................................... 41  
Working with Subrecords ....................................................................................................................... 47  
Record Filtering and Query ..................................................................................................................... 49  
   Listing All Record Instances ............................................................................................................... 49  
   Record Collection Filtering .................................................................................................................... 50  
   Executing SuiteSQL Queries Through REST Web Services .................................................................. 52  
   Working with SuiteAnalytics Datasets in REST Web Services ............................................................ 53  
   Collection Paging ............................................................................................................................... 55  
Error Handling and Logging in REST Web Services ................................................................................. 57  
   Error Handling in REST Web Services ............................................................................................... 57  
   Using the REST Web Services Execution Log ..................................................................................... 59
SuiteTalk REST Web Services Overview and Setup

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, 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.

The following tables provide an overview of fully supported and beta features in REST web services.

<table>
<thead>
<tr>
<th>Record API (...record/v1)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metadata – for more information, see Working with Resource Metadata.</td>
<td>Standard records – based on the record support flag in the REST API browser. You can see the support level of a record next to the record's name in the REST API Browser. For a list of fully supported records, you can also see the help topic REST Web Services Supported Records.</td>
</tr>
<tr>
<td>CRUD operations – for more information, see Working with Records.</td>
<td>Custom records – fully supported. For a list of usage notes, see Using CRUD Operations on Custom Records, Custom Lists, and Custom Transactions v1.</td>
</tr>
<tr>
<td>Record Collection Filtering – for more information, see Record Collection Filtering.</td>
<td></td>
</tr>
<tr>
<td>Record Actions – for more information, see Executing Record Actions.</td>
<td>Beta</td>
</tr>
<tr>
<td>Record Transformation – for more information, see Transforming Records.</td>
<td>Beta</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Query API (...query/v1)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workbook Dataset Execution – for more information, see Working with SuiteAnalytics Datasets in REST Web Services.</td>
<td>Beta</td>
</tr>
<tr>
<td>SuiteQL Query Execution – for more information, see Executing SuiteQL Queries Through REST Web Services.</td>
<td>Fully supported</td>
</tr>
</tbody>
</table>

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 REST Web Services

Consider the following information when working with REST web services.

- Enumeration values must be provided and are returned in the SuiteScript internal format. This is the format used by Record.getValue(options) and Record.setValue(options) 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.
- Translations in REST web services follow the same behavior as in SuiteScript. For custom records, a user’s preference (Home > Set Preferences > Language) is taken into the account. For standard records, translations are returned and updated in the company's primary language. Additionally, in the current version of REST web services, it is not possible to read and update translations on the translation sublist for both custom records and standard records. This behavior is different from the behavior of SuiteScript.

For more information, see the following topics:

- REST Web Services Key Concepts
- REST Web Services and Other Integration Options
- REST Web Services Prerequisites and Setup
- REST Web Services URL Schema and Account-Specific URLs

REST Web Services Key Concepts

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

HATEOAS

Hybermedia 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

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 RESTIets.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>REST Web Services</th>
<th>SOAP Web Services</th>
<th>RESTIets</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, OAuth 2.0)</td>
<td>Yes (user credentials; token-based authentication)</td>
<td>Yes (user credentials; token-based authentication, OAuth 2.0)</td>
</tr>
</tbody>
</table>

**Important:** User credentials are not recommended as an authentication method for SOAP web services. If you are currently using this method, you should transition to use Token-based Authentication instead.

**Important:** User credentials are not recommended as an authentication method for RESTIets. If you are currently using this method, you should transition to use either Token-based Authentication or OAuth 2.0 instead.

<table>
<thead>
<tr>
<th>Passing of Login Details</th>
<th>in OAuth authorization header</th>
<th>in body (SOAP)</th>
<th>in authorization header</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>
### REST Web Services and Other Integration Options

<table>
<thead>
<tr>
<th>Attribute</th>
<th>REST Web Services</th>
<th>SOAP Web Services</th>
<th>RESTIlets</th>
</tr>
</thead>
<tbody>
<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, suitable for mobile devices</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, Search</td>
<td>All functionality available through SuiteScript: CRUD, Search, 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</td>
</tr>
<tr>
<td>Performance</td>
<td>Using REST API, fewer calls may be required to accomplish a business flow. Therefore the overall performance may be better than SOAP and CSV.</td>
<td>SOAP web services require more calls to accomplish a business flow than the REST API.</td>
<td>RESTIlets are the fastest integration channel. All actions required for a business flow can be executed within a single call.</td>
</tr>
</tbody>
</table>

### REST Web Services Prerequisites and Setup

To use REST web services, the relevant features must be enabled in your account. Additionally, the REST web services user must have the required permissions assigned to the user’s role. See the following table for information about enabling the required features and assigning the required permissions.

<table>
<thead>
<tr>
<th>Feature or Permission</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>REST Web Services feature</td>
<td>Enable the feature at Setup &gt; Company &gt; Setup Tasks &gt; Enable Features, in the SuiteTalk (Web Services) section, on the SuiteCloud subtab. To use the feature, you must accept the SuiteCloud Terms of Service.</td>
</tr>
</tbody>
</table>
| REST Record Service (beta) feature  | Enable the feature at Setup > Company > Setup Tasks > Enable Features, in the SuiteTalk (Web Services) section, on the SuiteCloud subtab. To use the feature, you must accept the SuiteCloud Terms of Service.  
This feature enables you to work with the REST record service beta functionality. Using the REST record service, you can perform CRUD operations on beta records, filter record collections, interact with record metadata, and perform record actions and transformations. |
Feature or Permission | Usage
--- | ---
REST Query Service (beta) feature | Enable the feature at Setup > Company > Setup Tasks > Enable Features, in the SuiteTalk (Web Services) section, on the SuiteCloud subtab. To use the feature, you must accept the SuiteCloud Terms of Service.

This feature enables you to work with the REST query service beta functionality. Using the REST query service, you can work with datasets and related functionality.

SuiteAnalytics Workbook feature | Enable the feature at Setup > Company > Setup Tasks > Enable Features, on the Analytics subtab.

Permissions:
- REST Web Services
- Log in using Access Tokens
- SuiteAnalytics Workbook

See Standard Roles with the REST Web Services, SuiteAnalytics Workbook, and Log in Using Access Tokens Permissions for a list of roles that have the required permissions assigned by default.

See To assign the required permissions to a user’s role: for information about assigning the permissions manually.

**Note:** Using the administrator role for building web services integrations or accessing REST web services is not recommended for security reasons.

Standard Roles with the REST Web Services, SuiteAnalytics Workbook, and Log in Using Access Tokens Permissions

The following standard roles have the SuiteAnalytics Workbook, Log in using Access Tokens, and REST Web Services permissions assigned by default. For more information about standard roles, see the help topic NetSuite Roles Overview.

- Accountant
- Accountant (Reviewer)
- A/P Clerk
- A/R Clerk
- Bookkeeper
- CEO (Hands Off)
- CEO
- Sales Manager
- Sales Person
- Store Manager
- Support Manager
- Support Person
- Employee Center
- Warehouse Manager
- Payroll Manager
- Partner Center
- Intranet Manager
- Marketing Manager
- Marketing Assistant
If you create a new role, or if you use an existing custom role or a standard role that does not have the required permissions by default, you can assign the permissions manually.

**To assign the required permissions to a user’s 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 Setup.
8. In the Permission list, select Log in using Access Tokens.
9. In the Level list, select Full.
10. Click Add.
11. On the Permissions subtab, click Reports.
12. In the Permission list, select SuiteAnalytics Workbook.
13. In the Level list, select Edit.
14. Click Add.
15. 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 the following: <account ID>.<service>.netsuite.com. For example, if your account ID is 123456, your account-specific domain for REST web services is: 123456.suitetalk.api.netsuite.com

You can find the service URL for SuiteTalk (SOAP and REST web services) at Setup > Company > Setup Tasks > Company Information, on the Company URLs subtab. When you build a client application, you must ensure that the service URL is a configurable parameter.

⚠️ **Important:** The examples in this document use demo123 as the account ID. Substitute this with your own account ID.

You can access NetSuite resources through REST web services by using URLs specific to either record or query service.

A URL for a REST resource looks like the following.

<table>
<thead>
<tr>
<th>Example 1.1: Metadata Options for Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</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 API services endpoint
5. The specific services that is being accessed, for example, record, query, or other service
6. The service version, which specific to each service
7. The optional sub-service, for example, metadata-catalog, suiteql, or workbook
8. Query parameters

The following examples demonstrate the usage of the record and query services.

To use CRUD operations on records, you can use a URL similar to the following:

```
PATCH https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/42
```

For information about CRUD operations, see Working with Records.

Metadata-catalog is a subservice of the record service. To access resource metadata, you can use a URL similar to the following:

```
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/?select=customer
```

For information about using metadata, see Working with Resource Metadata.

Workbook is a subservice of the query service. To execute saved workbooks, you can use a URL similar to the following:

```
GET https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/workbook/custworkbook123/result
```
For information about saved datasets, see Working with SuiteAnalytics Datasets in REST Web Services.

SuiteQL is a subservice of the query service. To execute SuiteQL queries, you can use a URL similar to the following:

```
POST https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/suiteql?limit=10&offset=10
```

For information about SuiteQL queries in REST web services, see Executing SuiteQL Queries Through REST Web Services.
Authentication for REST Web Services

NetSuite supports two mechanisms that increase overall system security. Token-based authentication (TBA) and OAuth 2.0, a robust, industry standard-based mechanisms. These authentication mechanisms enable client applications to use a token to access NetSuite through APIs, eliminating the need for integrations to store user credentials.

REST web services do not support login through user credentials. For general information about TBA, see the help topic Token-based Authentication (TBA).

For general information about OAuth 2.0, see the help topic OAuth 2.0.

Setting Up Token-Based Authentication (TBA) for REST Web Services

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 Create 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.

After creating tokens, you use the token ID and token secret in your REST web services Postman environment. For information about setting up your environment, see Importing and Setting Up a Postman Environment.

For detailed information about using TBA in integration applications, see the help topic The Three-Step TBA Authorization Flow.

Setting Up OAuth 2.0 Authentication for REST Web Services

To set up OAuth 2.0, see the help topic Getting Started with OAuth 2.0.

To be able to use REST web services with OAuth 2.0, you must create an application using an integration record. See the help topic Create Integration Records for Applications to Use OAuth 2.0. As the last step of this procedure, make sure you note the client ID and client secret.

After you create the integration application, continue with a set up of an application for use with OAuth 2.0. For more information, see the help topic OAuth 2.0 for Integration Application Developers. After the application is set up, you can manage the authorized applications. For more information, see the help topic Managing OAuth 2.0 Authorized Applications.

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 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 API 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.

The 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 the Postman Collection
- Sending a Request From the Imported Collection
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. The template is located in the environment folder of the Postman environment template and collections archive.

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).

8. Enter the TBA credentials you created. For details about creating TBA credentials, see Authentication for REST Web Services.

9. Enter your company URL for SuiteTalk. You can find your company URL at Setup > Company > Setup Tasks > Company Information, on the Company URLs subtab.

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. Unzip the archive.

3. Click Import in the top menu of the Postman Application, and select the NetSuite REST API Tutorial.postman_collection.json file from the requests folder of the downloaded collection.

4. 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 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.

REST web services 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
- Working with JSON Schema Metadata
Getting Metadata

The endpoint for getting the metadata schema for all exposed records is http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog where services/rest is the name of the REST API endpoint, record is the name of the service we are trying to access, v1 is the service version, and metadata-catalog is the sub-resource, 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 http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog
```

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

```
{
  "items": [
    {
      "name": "account",
      "links": [
        {
          "rel": "canonical",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/account",
          "mediaType": "application/json"
        },
        {
          "rel": "alternate",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/account",
          "mediaType": "application/swagger+json"
        },
        {
          "rel": "alternate",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/account",
          "mediaType": "application/schema+json"
        },
        {
          "rel": "describes",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/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.

**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/services/rest/record/v1/metadata-catalog?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.
The following excerpt describes the record structure and the properties of different fields: their name, type, format, and attributes.

```
"salesOrder": {  
  "type": "object",  
  "properties": {  
    "code": {  
      "type": "integer",  
      "format": "int64",  
      "nullable": true  
    },  
    "startDate": {  
      "title": "Start Date",  
      "type": "string",  
      "description": "Sets the date for creating the first invoice. This should only be set if Advanced Billing is being used.",  
      "format": "date",  
      "nullable": true  
    },  
    "isDeleted": {  
      "type": "boolean"  
    },  
    "suppressedFromTitle": {  
      "type": "string",  
      "nullable": true  
    },  
    "clearedDate": {  
      "title": "Date Cleared",  
      "type": "string",  
      "format": "date",  
      "nullable": true  
    }  
  }  
}
```

The following excerpt describes a sublist of the sales order record.

```
"salesOrderItemRedemptionElement": {  
  "type": "object",  
  "properties": {  
    "links": {  
      "title": "Links",  
      "type": "array",  
      "readOnly": true,  
      "items": {  
        "$ref": "#/components/schemas/rsLink"  
      }  
    },  
    "authCodeApplied": {  
      "title": "Amount Applied",  
      "type": "number",  
      "format": "double",  
      "nullable": true  
    },  
    "authCodeRemaining": {  
      "title": "Available Credit",  
      "type": "number",  
      "format": "double",  
      "nullable": true  
    },  
    "giftCertificateAvailable": {  
      "type": "number",  
      "format": "double",  
      "nullable": true  
    },  
    "parentTransaction": {  
      "$ref": "#/components/schemas/salesOrder"  
    },  
    "authCode": {  
      "$ref": "#/components/schemas/nrResource"  
    }  
  }  
}
```

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:

![Swagger Editor Output](image)

**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.
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.

```
SET-HEADER Accept: application/schema+json
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/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.
Working with JSON Schema Metadata

```
{
  "type": "object",
  "properties": {
    "custentity_integer": {
      "title": "Integer Number",
      "type": "integer",
      "format": "int64",
      "nullable": true,
      "x-ns-custom-field": true
    },
    "startDate": {
      "title": "Start Date",
      "type": "string",
      "description": "Enter the date this person or company became a customer, lead or prospect.
If this person or company has a contract with you, enter the start date of the contract.
If you enter an estimate or an opportunity for this customer, this field will be updated with the date of that transaction.
",
      "format": "date",
      "nullable": true
    }
  },
  "$schema": "https://json-schema.org/draft-06/hyper-schema#"
}
```

**Note:** The JSON generated by 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.

```
{
  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.
Working with JSON Schema Metadata

Select schema: Empty schema

Input JSON:

```json
1 {  
2   "required": [  
3     "Currency",  
4     "entityId",  
5     "monthlyClosing"  
6   ],  
7   "type": "object",  
8   "properties": {  
9     "links": {  
10        "title": "links",  
11        "type": "array",  
12        "readOnly": true,  
13        "items": {  
14           "type": "object",  
15        }  
16     },  
17     "parent": {  
18        "title": "Parent",  
19        "type": "object",  
20     },  
21     "referrerList": {  
22        "title": "Referrer",  
23        "type": "object",  
24     },  
25     "startDate": {  
26        "title": "Start Date",  
27        "type": "string",  
28        "format": "date"  
29   }  
30 }
31
32 {  
33   "entityId": "My Customer",  
34   currency: { "id": "1" },  
35   monthlyClosing: "31 - End of the Month"
36}
```

✔ No errors found. JSON validates against the schema
Working with Records

Using REST Web Services for Custom Record Operations

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.

To work with records, you must have the REST Record Service (beta) feature enabled. For information about required features and permissions, see REST Web Services Prerequisites and Setup.

Note: The majority of SuiteScript records is available as beta records. Refer to the REST web services API and the developer documentation to learn the status of records.

- The REST API Browser
- NetSuite Record Structure
- Using CRUD Operations on Custom Records, Custom Lists, and Custom Transactions v1
- Creating a Record Instance
- Getting a Record Instance
- Updating a Record Instance
- Using the Upsert Operation
- Deleting a Record Instance
- Accessing Subresources in REST Web Services
- Using External IDs
- Using Datetime Fields
- Executing Record Actions
- Transforming Records
- Working with the Pricing Sublist on Item Records
- Using the REST Web Services SuiteScript Execution Context

The REST API Browser

Go to the REST API Browser.

The REST API Browser is a browser that provides a visual overview of the structure and capabilities of the REST web services Record API. The data presented in the REST API Browser is based on OpenAPI 3.0 metadata. For information about metadata, see Working with OpenAPI 3.0 Metadata.

The REST API browser provides the following information:

- The support level of records. Beta records are marked with a beta label.
- The summary of all records, sublists, schema definitions, and other objects.
- The available operations you can perform on a record.
- The description of URLs, HTTP methods, and request parameters used for CRUD operations.
The REST API Browser

- The structure of responses you can receive after performing an operation.
- The structure of error messages.
- The description of field names and field types, and the properties of fields.
- The subformat of strings associated with specialized fields such as date and time fields.

Navigating in the REST API Browser

The REST API Browser is designed to let you browse for records and schema definitions in an alphabetical order.

**To view records and schema definitions:**

1. Click the name of the appropriate record in the left pane.

![REST API Browser: Record API v1](https://[accountid].suitetalk.api.netsuite.com/services/rest/record/v1)

2. Review the path, the request parameters, and the responses.
3. Click the linked resource in the Responses section. By clicking the link, you can navigate to the schema definitions section of the REST API Browser.

![RESPONSES](application/vnd.oracle.resource+json, type=collection, application/vnd.oracle.resource+json, type=error)

4. In the schema definitions section, review the properties of the resource.
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.

Using CRUD Operations on Custom Records, Custom Lists, and Custom Transactions v1

REST web services enables you to perform CRUD (create, read, update, and delete) operations on custom record instances. You can use CRUD operations to perform business processing on custom records.

In NetSuite, you track all of the information in your account using records. Users with the Custom Record Entries permission can create custom records that collect information specific to the needs of their business.

You can also use all CRUD operations on custom list records and on basic, journal, and header only custom transactions. Sales and purchase custom transaction records are not supported. For information about custom transaction styles, see the help topic Custom Transaction Styles Overview.

You can use CRUD operations to perform business processing on custom lists and custom transactions.

For information about working with custom lists in the UI, see the help topic Custom Lists. For information about working with custom transactions in the UI, see the help topic Custom Transactions.

Usage Notes for Custom Records

Consider the following information when you perform CRUD operations on custom records.

- Creating custom record types through REST web services is not supported. To work with custom record definitions, you must have the Custom Records feature enabled on your account. To enable the feature, go to Setup > Company > Setup Tasks > Enable Features, and select Custom Records on the SuiteCloud subtab. You must create a custom record type in the UI, and then you can work with instances of the custom record type through REST web services. For information, see the help topic Creating Custom Record Types. You also can use SDF to create custom record types. For information, see the help topic Custom Record Types as XML Definitions.

- The record owner of a custom record can be changed through REST web services even if the Allow change option is not selected on the custom record. This behavior is the same as in SuiteScript.

- The ID field is always displayed through REST web services, even if the Show ID field is not set in the UI. This is standard behavior in non-UI channels for accessing NetSuite to make sure that the records can be identified.

- In parent–child record relationships, REST web services do not display children on the parent record. This behavior is different from the behavior of SuiteScript. You can use record collection filtering to access only child records that belong under a particular parent.
For information about using CRUD operations, see the following help topics.

- Creating a Record Instance
- Getting a Record Instance
- Updating a Record Instance
- Deleting a Record Instance

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.

REST web services do 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.

```json
POST https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/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/services/rest/record/v1/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: /services/rest/record/v1/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.

```json
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/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.
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

REST web services do not automatically expand sublists and subrecords. You can use the expandSubResources query parameter to expand sublists and subrecords. If the query parameter is not used, the response contains only the body fields of the record, and the sublists and subrecords are represented by links. See the example in Getting a Record Instance.

The following is an example where the query parameter is used.

```
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107?expandSubResources=true
```

In the response, note the expansion of the addressbookaddress subrecord.
Getting a Record Instance

```
    "href": "http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107/addressbook"
  },
  "items": [
    {
      "links": [
        {
          "rel": "self",
          "href": "http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107/addressbook/39"
        }
      ],
      "addressbookaddress": {
        "links": [
          {
            "rel": "self",
            "href": "http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107/addressbook/39/addressbookaddress"
          }
        ],
        "addr1": "417 Washington Blvd",
        "addressee": "Glenrock General Hospital",
        "addresseeText": "Alan Rath\nGlenrock General Hospital\n417 Washington Blvd\nGlenrock WY 82637",
        "attention": "Alan Rath",
        "city": "Glenrock",
        "country": "US",
        "dropdownstate": {
          "links": [
            {
              "rel": "self",
              "href": "http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/"
            }
          ],
          "id": "WY",
          "refName": "Wyoming"
        },
        "id": "152",
        "nKey": "152.0",
      }
    }
  ],
  "currency": {
    "links": [
      {
        "rel": "self",
        "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/currency/1"
      }
    ],
    "refName": "USD",
    "id": "1"
  }
```

Format of Selects and References

References to records in REST web services 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.

Format of Multiselects

REST web services 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.

```
"subsidiary": {
```
Getting a Record Instance

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

Format of Enumeration Values

REST web services uses **internal** values of enumeration values.

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.

```
PATCH https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/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/services/rest/record/v1/customer/107
```
Using the Upsert Operation

The upsert operation enables you to either create a record, 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.

```json
PUT http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/eid:CID002
{
  "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.

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

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

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

Accessing Subresources in REST Web Services

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. For more information about resources, see Resource.

Sublists and subrecords are subresources. Subresources can also be accessed through REST web services. Consider the following usage notes when working with subresources through REST web services.

Generally available and beta records only support the GET operation for subresources. POST, PATCH, and DELETE are not supported.

When working with records, you can perform the following operations on subresources:

Get a referenced record that is represented by a select field through the parent record.
Accessing Subresources in REST Web Services

GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/employee/107/subsidiary

Get a parent record's subresource: sublist.

GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/salesOrder/1/item
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/salesOrder/1/item/1

Get a parent record's subresource: subrecord.

GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/salesOrder/1/billingaddress

Get a parent record's subresource: sublist of a subrecord.

GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107/addressBook/39/addressBookAddress

Get a parent record's subresource: sublist of a subrecord of a sublist.

GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/salesOrder/6/item/1/inventoryDetail/inventoryAssignment/1

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/services/rest/record/v1/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/services/rest/record/v1/customer/eid:CID001/subscriptions/eid:SUID042
Using Datetime Fields

You can work with date and datetime fields using the date and datetime types in NetSuite. See the following sections for information about working with date and datetime types.

When setting a date field using REST web services, you must enter a date or datetime value. You can also specify a time zone in your request. Datetime values are returned in UTC format.

Types of Date-related Fields

The following four types of date-related fields are used in NetSuite.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date fields</td>
<td>10/05/2018</td>
<td>Date fields are used, for example, on custom records.</td>
</tr>
<tr>
<td>Time fields</td>
<td>10:15 am</td>
<td>Time fields are used, for example, on custom records.</td>
</tr>
<tr>
<td>Datetime fields</td>
<td>10/05/2018 10:15 am</td>
<td>Datetime fields are used, for example, on custom records.</td>
</tr>
<tr>
<td>Duration fields</td>
<td>25:45 (hours:minutes)</td>
<td>Duration fields are used in time tracking, for example, on time bill records.</td>
</tr>
</tbody>
</table>

Date Field

Date fields use the date element of datetime fields. For example: 2017-07-21. Date field values are not subject to any time zone conversion.

Time Fields

Time fields use the time element of datetime fields, with hours, minutes and second. For example: 17:32:28. Time field values are not subject to any time zone conversion.

Datetime Fields

REST web services use datetime fields in ISO 8601 format and in the UTC time zone. For example: 2017-07-21T17:32:28Z.

Date is the full-date notation as defined by RFC 3339, section 5.6, for example, 2017-07-21.

Date-time is the date-time notation as defined by RFC 3339, section 5.6, for example, 2017-07-21T17:32:28Z.

All datetime values loaded from NetSuite are converted to the UTC time. REST web services accepts datetime values in the same format and in the UTC time zone.

In REST web services, you can also specify the time zone in your request. For example, 2017-07-21T17:32:28+01:00.

Duration Fields

Duration fields use the same format as in UI, that is, "hh:mm". For example: "25:42". Duration field values are not subject to any time zone conversion.
Executing Record Actions

**Warning**: Record action execution 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. NetSuite warranties and product service levels shall not apply to the feature or to the impact of the feature on other portions of the NetSuite application. NetSuite 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. Please review [Beta Software Legal Notices](#). ORACLE CONFIDENTIAL. For authorized use only. Do not distribute to third parties.

REST web services support APIs that provide the programmatic equivalent of clicking a button in the NetSuite user interface. With the record action APIs, you can use REST web services to trigger the same business logic that is triggered by the click of a UI button. Record actions can increase productivity by automating regular tasks that previously had to be done manually in the UI.

By using record actions, you can update the state of a record. Approve and reject are two examples of record actions. When an approve or reject action is executed on a record, the approval status of the record is saved immediately.

REST web services support the same record actions supported by SuiteScript.

For a list of supported actions, see the help topic [Supported Record Actions](#).

Metadata is not provided for record actions.

You can call record actions using the HTTP POST method on the record instance. The action name is prefixed with the string `@`.

Action parameters can be passed in the request body in the JSON object.

The following is an example of a record action call with action parameters in the request body.

```json
POST http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/vendorPayment/3/@confirm
{
  "confirmationDate": "2019-1-31",
  "exchangeRate": "4.2",
  "postingPeriod": 348
}
```

You can also execute record actions without providing action parameters, as in the following example.

```json
POST http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/vendorPayment/3/@confirm
```

If an action is performed successfully, the HTTP 200 response is returned.

```json
HTTP/1.1 200 OK
Content-Type: application/vnd.oracle.resource+json; type=singular
{
  "links": [
    "self": "http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/vendorPayment/3/@confirm"
  ],
  "result": true
}
```

When a record action cannot be executed, an HTTP 4xx response is returned.

```json
HTTP/1.1 400 Bad Request
```
Transforming Records

**Warning:** Record transformation 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. NetSuite warranties and product service levels shall not apply to the feature or to the impact of the feature on other portions of the NetSuite application. NetSuite 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. Please review Beta Software Legal Notices. ORACLE CONFIDENTIAL. For authorized use only. Do not distribute to third parties.

In REST web services, you can transform a record from one type into another, using data from an existing records. For example, you can create an invoice record from an existing sales order record, using data from the sales order.

All record transformations available in SuiteScript are available in REST web services, too. Transformations are also supported in metadata. For information about the supported transformation types, see the help topic record.transform(options).

In REST web services, you use the POST method to send a record transformation request. The record transformation is executed in a single request. In the request URL, the following details must be specified:

- The record type you that you want to transform.
- The ID of the record that you want to transform.
- The transformation action. The transformation action must be formatted in the following way in the URL: `!transform`.
- The record type you that you want transform into.

In the following example, a sales order record is transformed into an invoice. If the transformation is performed successfully, an HTTP 204 – No Content response is returned.

```plaintext
POST http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/salesOrder/3/!transform/invoice
```

In the request body, you can also specify data for additional fields of the record, as shown in the following example.

```plaintext
{
    "memo": "REST, end date and bills date added",
    "enddate": "2020-01-14",
    "billdate": "2020-01-14",
    "item": {
        "items": [
            {
                "item": { "id": '3456' },
                "amount": 1
            }
        ]
    }
}
```
Working with the Pricing Sublist on Item Records

**Warning:** Support for pricing sublist data through 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. NetSuite warranties and product service levels shall not apply to the feature or to the impact of the feature on other portions of the NetSuite application. NetSuite 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. Please review Beta Software Legal Notices. ORACLE CONFIDENTIAL. For authorized use only. Do not distribute to third parties.

You can create, update, and access pricing sublist (pricing matrix) data through REST web services. The pricing sublist is available on item records, on the Sales/Pricing subtab.

The pricing sublist is supported if the following features are enabled:

- Multiple Currencies
- Multiple Prices
- Quantity Pricing

For each item, you can set multiple prices based on the following:

- Currency used for item prices. See the help topic Enabling the Multiple Currencies Feature.
- Price levels available for the item, as defined in the price level user-defined list. See the help topic Using Multiple Pricing.
- The quantity of items being sold, when the Quantity Pricing feature is enabled. See the help topic Using Quantity Pricing.

For general information about item pricing, see the following help topics:

- Item Pricing
- Setting Up Item Pricing
- Using Item Records

In REST web services, you can create or update price values on a per-row, per-column basis by using the POST operation. You can access the price values on a per-row, per-column basis by using the GET operation.

To create or update a pricing sublist, you send a POST request similar to the following and specify the pricing sublist data in the request body in an array of items:

```plaintext
POST https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/inventoryItem

"price": [
  {
    "currencyPage": { "id": "1" },
    "priceLevel": { "id": "1" },
    "quantity": { "value": 5 },
    "price": 25.5
  },
  ...
]
```
To access a pricing sublist, you send a GET request similar to the following:

```plaintext
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/inventoryitem/482/price?expandSubResources=true
```

### Using the REST Web Services SuiteScript Execution Context

Execution contexts provide information about how or when a script or workflow is triggered to execute. For example, a script can be triggered in response to an action in the NetSuite application or an action occurring in another context, such as a web services integration. You can use execution context filtering to ensure that your scripts or workflows are triggered only when necessary. This filtering can improve performance in contexts where the scripts or workflows are not required.

For NetSuite user event and client scripts, you can set up execution context filtering to control the contexts in which these scripts can run. You can set up execution context filtering in the Execution Context field on the Context Filtering subtab of the Script Deployment record. For information about script deployment, see the help topic [Methods of Deploying a Script](#). For more information about execution contexts in general, see the help topics [Execution Contexts](#) and `runtime.executionContext`.

If a script has its execution context set to REST web services, the script only executes if it is triggered through a REST web services request.

For example, you can deploy a script similar to the following on customer records. In this example, the `beforeSubmit` script is only executed on the record if the request is triggered through REST web services.

```javascript
define(['$record', '$runtime'],
    function(record, runtime) {
        function beforeSubmit(context) {
            log.debug("Before submit of "+ context.newRecord.type);
            var executionContext = runtime.executionContext;
            var isRestWS = (runtime.ContextType.REST_WS === executionContext);
            log.debug("Is originating from REST WS: "+ isRestWS);
        }
    })
```

After running the script, you can see its execution context in the script execution log. See the help topic [Using the Script Execution Log Tab](#). You can also find information about the execution context in the system notes of a record. See the help topic [Viewing System Notes on a Record](#).
Working with Sublists

Sublists are an important part of several NetSuite record types. In REST web services, 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.

**Note:** For information about the format of sublists in REST web services, see Format of Sublists and Subrecords.

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

```
{  "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": "inserted line 1" },        { "col": "inserted line 2" }      ]    }  }}
```

Response – Post State

Response:

201 Created

Location: .../myrecord/101

```
{  "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": [      ]    }  }}
```
Creating a Sublist

```
{ "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

```
{ "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" : {
"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" },
}
Request Body – Nullifying a Sublist

PATCH .../myrecord/100

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

Response – Post State

Response:

204 No Content

Post State:

```
{  
  "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
Replacing a Sublist

All 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

```json
{
  "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": "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": {
    "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

```json
{
  "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" }
    ]
  }
}
```
Replacing a Sublist

```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": {
    "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

```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" }
      ]
    }
  }
}
```
Replacing a Sublist

Response – Post State

Response:

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" }
        ]
    },
    "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
Replacing a Sublist

Post State:

```json
{
  "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": "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

```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:

204 No Content

Post State:

```json
{
  "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" }
    ]
  }
}
```
Replacing a Sublist

SuiteTalk REST Web Services
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.

Note: For information about the format of subrecords in REST web services, see Format of Sublists and Subrecords.

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.

```json
POST http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/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.

```json
GET http://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/42/addressbook/24/addressbookaddress
{
  "addressee": "Dwight Schrute",
  "city": "New York",
  "country": "US",
  "dropdownstate": {
```
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.

REST web services only supports limited record query. Joins are not supported.

To work with record filtering and queries, you must have the REST Query Service (beta) feature enabled. For information about required features and permissions, see REST Web Services Prerequisites and Setup.

For more information, see the following topics:

- Listing All Record Instances
- Record Collection Filtering
- Executing SuiteQL Queries Through REST Web Services
- Working with SuiteAnalytics Datasets in REST Web Services
- 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 `.../services/rest/record/v1/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.
The following is a shortened example of a response.

```json
{
  "links": [
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer?limit=10000&offset=0"
    }
  ],
  "items": [
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107"
        }
      ],
      "id": "107"
    },
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/41"
        }
      ],
      "id": "41"
    },
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/90"
        }
      ],
      "id": "90"
    }
  ],
  "totalResults": 3
}
```

### 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, an operator, and a value. You can join several conditions using the AND / OR logical operators, and you can use () to mark precedence.

The following table contains the list of available query operators with their associated field types.

<table>
<thead>
<tr>
<th>Field Type</th>
<th>Allowed Filters</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>EMPTY, EMPTY_NOT</td>
</tr>
<tr>
<td>Boolean</td>
<td>IS, IS_NOT</td>
</tr>
<tr>
<td>Double, Integer, Float, Number, Duration</td>
<td>ANY_OF, ANY_OF_NOT, BETWEEN, BETWEEN_NOT, EQUAL, EQUAL_NOT, GREATER, GREATER_NOT, GREATER_OR_EQUAL, GREATER_OR_EQUAL_NOT, LESS, LESS_NOT, LESS_OR_EQUAL, LESS_OR_EQUAL_NOT, WITHIN, WITHIN_NOT</td>
</tr>
<tr>
<td>String</td>
<td>CONTAIN, CONTAIN_NOT, IS, IS_NOT, START_WITH, START_WITH_NOT, END_WITH, END_WITH_NOT</td>
</tr>
</tbody>
</table>
### Field Type | Allowed Filters
---|---
Date / Time | AFTER, AFTER_NOT, BEFORE, BEFORE_NOT, ON, ON_NOT, ON_OR_AFTER, ON_OR_AFTER_NOT, ON_OR_BEFORE, ON_OR_BEFORE_NOT

Not all operators accept one value. Some operators do not require any value, some operators require two values, and some operators accept any number of values. Consider the following examples:

- **Unary operators**: The EMPTY and EMPTY_NOT operators do not accept any values. For example: `q=companyName EMPTY`
- **Ternary operators**: The BETWEEN, BETWEEN_NOT, WITHIN, and WITHIN_NOT operators accept two values. For example: `q=id BETWEEN_NOT [1, 42]`
- **N-ary operators**: The ANY_OF and ANY_OF_NOT operators do accept one or any higher number of values. For example: `q=id ANY_OF [1, 2, 3, 4, 5]`

You can find the field available for filtering in the metadata. For information about metadata, see [Working with Resource Metadata](#).

For information about using datetime fields in queries, see [Using Datetime Fields](#).

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.

```plaintext
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/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:

```json
{
  "links": [
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer?limit=1000&offset=0"
    }
  ],
  "items": [
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107"
        }
      ],
      "id": "107"
    }
  ],
  "totalResults": 1
}
```

When your condition value 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 value):**
  
  ```plaintext
  GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer?q=companyname START_WITH "Another Company"
  ```

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

- Find customers created in 2019 (date value, AND operator):
  GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/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):
  GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer?q=creditlimit GREATER_OR_EQUAL 1000 OR creditlimit LESS_OR_EQUAL 10

**Note:** When you join more than one conditions, the AND logical operator has priority over the OR logical operator. To change this behavior, use parentheses as follows: "(‘ and ‘)".

### Executing SuiteQL Queries Through REST Web Services

SuiteQL is a query language based on the SQL database query language. SuiteQL provides advanced dynamic query capabilities that can be used to access NetSuite records.

For more information about SuiteQL in general, see the help topic [SuiteQL](#).

You can execute SuiteQL queries through REST web services by sending a POST request to the suiteql resource, and specifying the query in the request body after the body parameter q.

In the request URL, you can also specify the number of results you want to return in a single page and the page offset. For information about paging and offset values, see [Collection Paging](#).

You can use the Analytics Browser to get information about all record types and fields that are available through the analytics data source. The browser includes a page for each record type with all available fields and joined record types, if applicable. For each field, you can also find whether that field is available for use in SuiteAnalytics Workbook, SuiteScript, SuiteTalk REST web services, and SuiteAnalytics Connect. For more information, see the help topic [Analytics Browser](#).

Using SuiteQL queries, you can return a maximum of 100,000 results. For more information, see the help topic [query.runSuiteQLPaged(options)](#).

The following example shows a SuiteQL query executed through REST web services. Note that Prefer: transient is a required header parameter.

```plaintext
POST https://demo123.suitetalk.api.netsuite.com/services/rest/query/v1/suiteql?limit=10&offset=10
Prefer: transient
{
  "q": "SELECT email, COUNT(*) as count FROM transaction GROUP BY email"
}
```

The following is a shortened response.

```plaintext
{
  "links": ..., 
  "count": 3, 
  "offset": 10, 
  "totalResults": 53, 
  "items": [
    {
      "links": [],
      "email": "test@netsuite.com", 
      "count": "143"
    }
  ]
}
```
In SuiteQL queries, you can also use advanced SQL functions, for example joins and concatenations. The following example shows a request where the first and last names from employee records are concatenated as full name.

```json
{
  "q": "SELECT CONCAT(firstname,lastname) as fullname FROM employee"
}
```

When working with SuiteQL, consider the limitation listed at SuiteQL Limitations and Exceptions.

## Working with SuiteAnalytics Datasets in REST Web Services

**Warning:** Support for dataset execution and getting lists of datasets 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. NetSuite warranties and product service levels shall not apply to the feature or to the impact of the feature on other portions of the NetSuite application. NetSuite 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. Please review Beta Software Legal Notices. ORACLE CONFIDENTIAL. For authorized use only. Do not distribute to third parties.

In SuiteAnalytics Workbook, you analyze your company data using two distinct objects: a dataset and a workbook.

Datasets are the basis for all workbooks and workbook components in your account. In a dataset, you combine record type fields and criteria filters to create a query. The results of this query act as the source data for your workbooks.

Prior to 2020.1, you defined, visualized, and analyzed SuiteAnalytics Workbook source data in a single workbook object.

As of 2020.1, workbooks and datasets exist as separate objects, each with their own respective owners and accessibility rights. You define a dataset by selecting record types, fields, and criteria.

REST web services support the execution of datasets. Dataset execution replaces the previously available workbook execution function. You can execute both standard and custom datasets by using the dataset ID in your GET request.

When you work with datasets in REST web services, consider the following:

- You must have the SuiteAnalytics Workbook feature enabled. See REST Web Services Prerequisites and Setup.
The SuiteAnalytics Workbook permission must be assigned to the user's role. See REST Web Services Prerequisites and Setup.

Datasets must be created in the SuiteAnalytics Workbook UI and cannot be created through REST web services. For information about creating datasets, see the help topic Defining a Dataset.

Metadata is not provided for datasets.

Executing Datasets Through REST Web Services

To execute a dataset, you send a GET request similar to the following:

```
GET https://demo123.suitetalk.api.netuite.com/services/rest/query/v1/dataset/ExampleDataset1/result
```

The following is the response. The response is in a key-value structure or table.

```
{
  "links": [
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netuite.com/services/rest/query/v1/dataset/ExampleDataset1/result?limit=1000&offset=0"
    }
  ],
  "items": [
    {
      "links": [],
      "amount": "5.33333334",
      "amount2": "2",
      "entity": "Customer AU",
      "item": "Item Tax 2 Lbs(AU)",
      "itemcount": "1",
      "postingperiod": "May 2010",
      "trandate": "5/28/2010"
    }
  ],
  "count": 1,
  "hasMore": false,
  "offset": 0,
  "totalResults": 1
}
```

When executing datasets through REST web services, consider the following:

- The response from a dataset execution returned through REST web services may have different format and values than responses for record instance requests.
- A REST web services request executes a single dataset. You cannot execute multiple datasets through REST web services at the same time.

Getting a List of Datasets Through REST Web Services

You can list all existing SuiteAnalytics Workbook datasets through REST web services. Datasets must be created in the SuiteAnalytics Workbook UI and cannot be created through REST web services. For information about creating datasets, see the help topic Defining a Dataset.

To list all existing datasets, you send a GET request similar to the following:

```
GET https://demo123.suitetalk.api.netuite.com/services/rest/query/v1/dataset/
```
The result contains the name, ID, and description of the datasets, and it provides information about the record type the dataset works on.

You cannot use collection paging when you list all datasets. A request for a list of datasets returns all datasets on a single page.

**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:

```plaintext
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer?limit=2
```

The following is an example of a response:

```json
{
  "links": [
    {
      "rel": "next",
      "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/account?limit=2&offset=2"
    },
    {
      "rel": "last",
      "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/account?limit=22&offset=2"
    },
    {
      "rel": "self",
      "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer?limit=2&offset=0"
    }
  ],
  "items": [
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/107"
        }
      ],
      "id": "107"
    },
    {
      "links": [
        {
          "rel": "self",
          "href": "https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/41"
        }
      ],
      "id": "41"
    }
  ],
  "count": 2,
  "offset": 0,
  "hasMore": true,
  "totalResults": 3
}
```

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:

```xml
GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/account?limit=2&offset=2
```

The following is an example of a response:

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

For information about error handling and logging, see the following topics:

- Error Handling in REST Web Services
- Using the REST Web Services Execution Log

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 error messages clarify whether an error occurred in the request URL, the request headers, the query parameters, or the request body. Error messages also provide more information about the cause of errors and provide possible solutions. Error responses contain the following fields:

- status – the HTTP status code.
- errorDetails – an array with error details. One error response can contain multiple error details.
  - detail – human-readable error description.
  - errorCode – the error category in a machine-friendly format.
  - errorPath (optional) – a machine-readable JSON path that points to the request body where the error occurred.
  - urlPath (optional) – a machine-readable URI path that points to the part of the URL where the error occurred.
  - errorHeader (optional) – the machine-friendly name of the request header where the error occurred.
  - errorQueryParam (optional) – the machine-friendly name of the request query parameter where the error occurred.

The following sections show some common errors with their descriptions.

Invalid Login Information

The following error is returned if the request contains an invalid login token.

```json
{
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
  "title": "Unauthorized",
  "status": 401,
  "errorDetails": [
    {
      "detail": "Invalid login attempt. For more details, see the Login Audit Trail in the NetSuite UI at Setup > Users/Roles > User Management > View Login Audit Trail."
    }
  ]
}
```
Request with an Invalid Body

The following is a POST request https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/salesOrder with an invalid reference to a sublist item in the request body.

```
{  
  "entity": { "id": 187  },  
  "location": { "id": 1  },  
  "item": {  
    "items": [
      {  
        "item": { "id": 9999  },  
        "amount": 1  
      }  
    ]  
  }  
}
```

The request returns the following error with the JSON path o:errorCode pointing to the invalid value.

```
{  
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.4.1",  
  "title": "Bad Request",  
  "status": 400,  
  "o:errorCode": "INVALID_CONTENT",  
  "o:errorPath": "item.items[0].item"  
}
```

Request for a Non-existent Record

The request GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/customer/999 returns the following error if the request is for a non-existent customer.

```
{  
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",  
  "title": "Not Found",  
  "status": 404,  
  "o:errorCode": "NONEXISTENT_ID"  
}
```

Invalid Request

The request GET https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog/customer, salesOrder returns the following error because there is an extra space in the query

SuiteTalk REST Web Services
parameters. The correct request is https://demo123.suitetalk.api.netsuite.com/services/rest/record/v1/metadata-catalog?select=customer,salesOrder.

```json
{
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
  "title": "Bad Request",
  "status": 400,
  "errorDetails": [
    {
      "detail": "The request could not be understood by the server due to malformed syntax."
    }
  ]
}
```

### 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](#).

If this error occurs, retry sending the request. For information about implementing a retry logic in your code, see the help topic [Retrying Failed Web Services Requests](#).

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

### 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.

```json
{
  "type": "https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html",
  "title": "Internal Server Error",
  "status": 500,
  "errorDetails": [
    {
      "detail": "An unexpected error occurred. Error ID: jrgbpyyplphihshmlxyt",
      "errorCode": "UNEXPECTED_ERROR"
    }
  ]
}
```

### Using the REST Web Services Execution Log

Each integration record includes a subtab labeled REST Web Services, under the Execution Log subtab. This log lists REST web services requests that are uniquely identified with the integration record you
are currently viewing. That is, the log includes those requests that referenced the integration record's consumer key.

The execution log includes information about all requests and responses.

To see the execution log for a REST web services integration:

1. Go to Setup > Integration > Manage Integrations.
2. Select an integration record from the list.

For each request, the log includes the following details:

- The date and time that the request was made.
- The duration of the request.
- The status of the request.
- The email address of the user who sent the request.
- The HTTP method used and the HTTP status code
- The URL to the record or record type.
- The request and response body.

**Note:** The requests and responses logged in the execution log contain the values of sensitive fields in masked out format. For information about using encrypted format on custom fields, see the help topic *Encrypted Custom Field Stored Values*.