Document

JSforce library document with brief usage examples of each API

Connection

Username and Password Login

When you have a Salesforce username and password (and maybe security token, if required), you can use `Connection#login(username, password)` (/jsforce/doc/Connection.html#login) to establish a connection to Salesforce.

By default, it uses SOAP login API (so no OAuth2 client information is required).

```
var jsforce = require('jsforce');
var conn = new jsforce.Connection({
  // you can change loginUrl to connect to sandbox or prerelease env.
  // loginUrl : 'https://test.salesforce.com'
});
conn.login(username, password, function(err, userInfo) {
  if (err) { return console.error(err); }
  // Now you can get the access token and instance URL information.
  // Save them to establish connection next time.
  console.log(conn.accessToken);
  console.log(conn.instanceUrl);
  // logged in user property
  console.log("User ID: " + userInfo.id);
  console.log("Org ID: " + userInfo.organizationId);
  // ...
});
```

Username and Password Login (OAuth2 Resource Owner Password Credential)

When OAuth2 client information is given, `Connection#login(username, password + security_token)` (/jsforce/doc/Connection.html#login) uses OAuth2 Resource Owner Password Credential flow to login to Salesforce.

```
var jsforce = require('jsforce');
var conn = new jsforce.Connection({
  oauth2 : {
    // you can change loginUrl to connect to sandbox or prerelease env.
    // loginUrl : 'https://test.salesforce.com',
    clientId : '<your Salesforce OAuth2 client ID is here>',
    clientSecret : '<your Salesforce OAuth2 client secret is here>',
    redirectUri : '<callback URI is here>'
  }
});
conn.login(username, password, function(err, userInfo) {
  if (err) { return console.error(err); }
  // Now you can get the access token and instance URL information.
  // Save them to establish connection next time.
  console.log(conn.accessToken);
  console.log(conn.instanceUrl);
  // logged in user property
  console.log("User ID: " + userInfo.id);
  console.log("Org ID: " + userInfo.organizationId);
  // ...
});
```

Session ID

If Salesforce session ID and its server URL information is passed from Salesforce (from 'Custom Link' or something), you can pass it to the constructor.

```
var jsforce = require('jsforce');
var conn = new jsforce.Connection({
  serverUrl : '<your Salesforce server URL (e.g. https://na1.salesforce.com) is here>',
  sessionId : '<your Salesforce session ID is here>'
});
```

Access Token

After the login API call or OAuth2 authorization, you can get the Salesforce access token and its instance URL. Next time you can use them to establish a connection.

```
var jsforce = require('jsforce');
var conn = new jsforce.Connection({
  instanceUrl : '<your Salesforce server URL (e.g. https://na1.salesforce.com) is here>',
  accessToken : '<your Salesforrce OAuth2 access token is here>'
});
```

Access Token with Refresh Token

If a refresh token is provided in the constructor, the connection will automatically refresh the access token when it has expired.

NOTE: Refresh token is only available for OAuth2 authorization code flow.

```
var jsforce = require('jsforce');
var conn = new jsforce.Connection({
  oauth2 : {
    clientId : '<your Salesforce OAuth2 client ID is here>',
    clientSecret : '<your Salesforce OAuth2 client secret is here>',
    redirectUri : '<your Salesforce OAuth2 redirect URI is here>'
  },
  instanceUrl : '<your Salesforce server URL (e.g. https://na1.salesforce.com) is here>',
  accessToken : '<your Salesforrce OAuth2 access token is here>',
  refreshToken : '<your Salesforce OAuth2 refresh token is here>'
});
conn.on("refresh", function(accessToken, res) {
  // Refresh event will be fired when renewed access token
  // to store it in your storage for next request
});

// Alternatively, you can use the callback style request to fetch the refresh token
conn.oauth2.refreshToken(refreshToken, (err, results) => {
  if (err) return reject(err);
  resolve(results);
});
```

Logout

Call `Connection#logout()` (/jsforce/doc/Connection.html#logout) to logout from the server and invalidate current session. It is valid for both SOAP API based sessions and OAuth2 based sessions.

```
var jsforce = require('jsforce');
var conn = new jsforce.Connection({
  sessionId : '<session id to logout>',
  serverUrl : '<your Salesforce Server url to logout>'
});
conn.logout(function(err) {
  if (err) { return console.error(err); }
  // now the session has been expired.
});
```

OAuth2

(Following examples are assuming running on express.js framework.)

Authorization Request

First, you should redirect user to Salesforce page to get authorized. You can get Salesforce authorization page URL by `OAuth2#getAuthorizationUrl(options)` (/jsforce/doc/OAuth2.html#getAuthorizationUrl).

```
var jsforce = require('jsforce');
//
// OAuth2 client information can be shared with multiple connections.
//
var oauth2 = new jsforce.OAuth2({
  // you can change loginUrl to connect to sandbox or prerelease env.
  // loginUrl : 'https://test.salesforce.com',
  clientId : '<your Salesforce OAuth2 client ID is here>',
  clientSecret : '<your Salesforce OAuth2 client secret is here>',
  redirectUri : '<callback URI is here>'
});
//
// Get authorization url and redirect to it.
//
app.get('/oauth2/auth', function(req, res) {
  res.redirect(oauth2.getAuthorizationUrl({ scope : 'api id web' }));
});
```

Access Token Request

After the acceptance of authorization request, your app is callbacked from Salesforce with authorization code in URL parameter. Pass the code to `Connection#authorize(code)` (/jsforce/doc/Connection.html#authorize) and get access token.

For the refresh token to be returned from Salesforce, make sure that the following Scope is included in the Connected App `Perform requests on your behalf at any time (refresh_token, offline_access)` and `refresh_token` is included in the call to `getAuthorizationUrl()`.

```
//
// Pass received authorization code and get access token
//
app.get('/oauth2/callback', function(req, res) {
  var conn = new jsforce.Connection({ oauth2 : oauth2 });
  var code = req.param('code');
  conn.authorize(code, function(err, userInfo) {
    if (err) { return console.error(err); }
    // Now you can get the access token, refresh token, and instance URL information.
    // Save them to establish connection next time.
    console.log(conn.accessToken);
    console.log(conn.refreshToken);
    console.log(conn.instanceUrl);
    console.log("User ID: " + userInfo.id);
    console.log("Org ID: " + userInfo.organizationId);
    // ...
    res.send('success'); // or your desired response
  });
});
```

Query

Using SOQL

By using `Connection#query(soql)` (/jsforce/doc/Connection.html#query), you can achieve very basic SOQL query to fetch Salesforce records.

```
var records = [];
conn.query("SELECT Id, Name FROM Account", function(err, result) {
  if (err) { return console.error(err); }
  console.log("total : " + result.totalSize);
  console.log("fetched : " + result.records.length);
});
```

Edit Run

Callback Style

There are two ways to retrieve the result records.

As we have seen above, our package provides widely-used callback style API call for query execution. It returns one API call result in its callback.

```
var records = [];
conn.query("SELECT Id, Name FROM Account", function(err, result) {
  if (err) { return console.error(err); }
  console.log("total : " + result.totalSize);
  console.log("fetched : " + result.records.length);
  console.log("done ? : " + result.done);
  if (!result.done) {
    // you can use the locator to fetch next records set.
    // Connection#queryMore()
    console.log("next records URL : " + result.nextRecordsUrl);
  }
});
```

Edit Run

Event-Driven Style

When a query is executed, it emits "record" event for each fetched record. By listening the event you can collect fetched records.

If you want to fetch records exceeding the limit number of returning records per one query, you can use `autoFetch` option in `Query#execute(options)` (/jsforce/doc/Query.html#execute) (or its synonym `Query#exec(options)` (/jsforce/doc/Query.html#exec), `Query#run(options)` (/jsforce/doc/Query.html#run)) method. It is recommended to use `maxFetch` option also, if you have no idea how large the query result will become.

When query is completed, `end` event will be fired. The `error` event occurs something wrong when doing query.

```
var records = [];
var query = conn.query("SELECT Id, Name FROM Account")
  .on("record", function(record) {
    records.push(record);
  })
  .on("end", function() {
    console.log("total in database : " + query.totalSize);
    console.log("total fetched : " + query.totalFetched);
  })
  .on("error", function(err) {
    console.error(err);
  })
  .run({ autoFetch : true, maxFetch : 4000 }); // synonym of Query#execute();
```

Edit Run

NOTE: When `maxFetch` option is not set, the default value (10,000) is applied. If you really want to fetch more records than the default value, you should explicitly set the maxFetch value in your query.

NOTE: In ver. 1.2 or earlier, the callback style (or promise style) query invokation with `autoFetch` option only returns records in first fetch. From 1.3, it returns all records retrieved up to `maxFetch` value.

Using Query Method-Chain

Basic Method Chaining

By using `SObject#find(conditions, fields)` (/jsforce/doc/SObject.html#find), you can do query in JSON-based condition expression (like MongoDB). By chaining other query construction methods, you can create a query programatically.

```
//
// Following query is equivalent to this SOQL
//
// "SELECT Id, Name, CreatedDate FROM Contact
//  WHERE LastName LIKE 'A%' AND CreatedDate >= YESTERDAY AND Account.Name = 'Sony, Inc.'
//  ORDER BY CreatedDate DESC, Name ASC
//  LIMIT 5 OFFSET 10"
//
conn.sobject("Contact")
  .find(
    // conditions in JSON object
    { LastName : { $like : 'A%' },
      CreatedDate: { $gte : jsforce.Date.YESTERDAY },
      'Account.Name' : 'Sony, Inc.' },
    // fields in JSON object
    { Id: 1,
      Name: 1,
      CreatedDate: 1 }
  )
  .sort({ CreatedDate: -1, Name : 1 })
  .limit(5)
  .skip(10)
  .execute(function(err, records) {
    if (err) { return console.error(err); }
    console.log("fetched : " + records.length);
  });
```

Edit Run

Another representation of the query above.

```
conn.sobject("Contact")
  .find({
    LastName : { $like : 'A%' },
    CreatedDate: { $gte : jsforce.Date.YESTERDAY },
    'Account.Name' : 'Sony, Inc.'
  },
    'Id, Name, CreatedDate' // fields can be string of comma-separated field names
                            // or array of field names (e.g. [ 'Id', 'Name', 'CreatedDate' ])
  )
  .sort('-CreatedDate Name') // if "-" is prefixed to field name, considered as descending.
  .limit(5)
  .skip(10)
  .execute(function(err, records) {
    if (err) { return console.error(err); }
    console.log("record length = " + records.length);
    for (var i=0; i<records.length; i++) {
      var record = records[i];
      console.log("Name: " + record.Name);
      console.log("Created Date: " + record.CreatedDate);
    }
  });
```

Edit Run

Wildcard Fields

When `fields` argument is omitted in `SObject#find(conditions, fields)` (/jsforce/doc/SObject.html#find) call, it will implicitly describe current SObject fields before the query (lookup cached result first, if available) and then fetch all fields defined in the SObject.

NOTE: In the version less than 0.6, it fetches only `Id` field if `fields` argument is omitted.

```
conn.sobject("Contact")
  .find({ CreatedDate: jsforce.Date.TODAY }) // "fields" argument is omitted
  .execute(function(err, records) {
    if (err) { return console.error(err); }
    console.log(records);
  });
```

Edit Run

The above query is equivalent to:

```
conn.sobject("Contact")
  .find({ CreatedDate: jsforce.Date.TODAY }, '*') // fields in asterisk, means wildcard.
  .execute(function(err, records) {
    if (err) { return console.error(err); }
    console.log(records);
  });
```

Edit Run

Query can also be represented in more SQL-like verbs - `SObject#select(fields)` (/jsforce/doc/SObject.html#select), `Query#where(conditions)` (/jsforce/doc/Query.html#where), `Query#orderby(sort, dir)` (/jsforce/doc/Query.html#orderby), and `Query#offset(num)` (/jsforce/doc/Query.html#offset).

```
conn.sobject("Contact")
  .select('*, Account.*') // asterisk means all fields in specified level are targeted.
  .where("CreatedDate = TODAY") // conditions in raw SOQL where clause.
  .limit(10)
  .offset(20) // synonym of "skip"
  .execute(function(err, records) {
    for (var i=0; i<records.length; i++) {
      var record = records[i];
      console.log("First Name: " + record.FirstName);
      console.log("Last Name: " + record.LastName);
      // fields in Account relationship are fetched
      console.log("Account Name: " + record.Account.Name);
    }
  });
```

Edit Run

You can also include child relationship records into query result by calling `Query#include(childRelName)` (/jsforce/doc/Query.html#include). After `Query#include(childRelName)` (/jsforce/doc/Query.html#include) call, it enters into the context of child query. In child query context, query construction call is applied to the child query. Use `SubQuery#end()` (/jsforce/doc/SubQuery.html#end) to recover from the child context.

```
//
// Following query is equivalent to this SOQL
//
// "SELECT Id, FirstName, LastName, ...,
//         Account.Id, Acount.Name, ...,
//         (SELECT Id, Subject, … FROM Cases
//          WHERE Status = 'New' AND OwnerId = :conn.userInfo.id
//          ORDER BY CreatedDate DESC)
//  FROM Contact
//  WHERE CreatedDate = TODAY
//  LIMIT 10 OFFSET 20"
//
conn.sobject("Contact")
  .select('*, Account.*')
  .include("Cases") // include child relationship records in query result.
     // after include() call, entering into the context of child query.
     .select("*")
     .where({
        Status: 'New',
        OwnerId : conn.userInfo.id,
     })
     .orderby("CreatedDate", "DESC")
     .end() // be sure to call end() to exit child query context
  .where("CreatedDate = TODAY")
  .limit(10)
  .offset(20)
  .execute(function(err, records) {
    if (err) { return console.error(err); }
    console.log('records length = ' + records.length);
    for (var i=0; i<records.length; i++) {
      var record = records[i];
      console.log("First Name: " + record.FirstName);
      console.log("Last Name: " + record.LastName);
      // fields in Account relationship are fetched
      console.log("Account Name: " + record.Account.Name);
      //
      if (record.Cases) {
        console.log("Cases total: " + record.Cases.totalSize);
        console.log("Cases fetched: " + record.Cases.records.length);
      }
    }
  });
```

Edit Run

Search

`Connection#search` enables you to search records with SOSL in multiple objects.

```
conn.search("FIND {Un*} IN ALL FIELDS RETURNING Account(Id, Name), Lead(Id, Name)",
  function(err, res) {
    if (err) { return console.error(err); }
    console.log(res);
  }
);
```

Edit Run

CRUD

JSforce supports basic "CRUD" operation for records in Salesforce. It also supports multiple record manipulation, but it consumes one API request per record. Be careful for the API quota consumption.

Retrieve

`SObject#retrieve(id)` (/jsforce/doc/SObject.html#retrieve) fetches a record or records specified by id(s) in first argument.

```
// Single record retrieval
conn.sobject("Account").retrieve("0017000000hOMChAAO", function(err, account) {
  if (err) { return console.error(err); }
  console.log("Name : " + account.Name);
  // ...
});
```

Edit Run

```
// Multiple record retrieval
conn.sobject("Account").retrieve([
  "0017000000hOMChAAO",
  "0017000000iKOZTAA4"
], function(err, accounts) {
  if (err) { return console.error(err); }
  for (var i=0; i < accounts.length; i++) {
    console.log("Name : " + accounts[i].Name);
  }
  // ...
});
```

Edit Run

Create

`SObject#create(record)` (/jsforce/doc/SObject.html#create) (or its synonym `SObject#insert(record)` (/jsforce/doc/SObject.html#insert)) creates a record or records given in first argument.

```
// Single record creation
conn.sobject("Account").create({ Name : 'My Account #1' }, function(err, ret) {
  if (err || !ret.success) { return console.error(err, ret); }
  console.log("Created record id : " + ret.id);
  // ...
});
```

Edit Run

```
// Multiple records creation
conn.sobject("Account").create([
  { Name : 'My Account #1' },
  { Name : 'My Account #2' }
],
function(err, rets) {
  if (err) { return console.error(err); }
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("Created record id : " + rets[i].id);
    }
  }
  // ...
});
```

Edit Run

Update

`SObject#update(record)` (/jsforce/doc/SObject.html#update) updates a record or records given in first argument.

```
// Single record update
conn.sobject("Account").update({
  Id : '0017000000hOMChAAO',
  Name : 'Updated Account #1'
}, function(err, ret) {
  if (err || !ret.success) { return console.error(err, ret); }
  console.log('Updated Successfully : ' + ret.id);
  // ...
});
```

Edit Run

```
// Multiple records update
conn.sobject("Account").update([
  { Id : '0017000000hOMChAAO', Name : 'Updated Account #1' },
  { Id : '0017000000iKOZTAA4', Name : 'Updated Account #2' }
],
function(err, rets) {
  if (err) { return console.error(err); }
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("Updated Successfully : " + rets[i].id);
    }
  }
});
```

Edit Run

Delete

`SObject#destroy(id)` (/jsforce/doc/SObject.html#destroy) (or its synonym `SObject#del(id)` (/jsforce/doc/SObject.html#del), `SObject#delete(id)` (/jsforce/doc/SObject.html#delete)) deletes a record or records given in first argument.

```
// Single record deletion
conn.sobject("Account").destroy('0017000000hOMChAAO', function(err, ret) {
  if (err || !ret.success) { return console.error(err, ret); }
  console.log('Deleted Successfully : ' + ret.id);
});
```

Edit Run

```
// Multiple records deletion
conn.sobject("Account").del([ // synonym of "destroy"
  '0017000000hOMChAAO',
  '0017000000iKOZTAA4'
],
function(err, rets) {
  if (err) { return console.error(err); }
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("Deleted Successfully : " + rets[i].id);
    }
  }
});
```

Edit Run

Upsert

`SObject#upsert(record, extIdField)` (/jsforce/doc/SObject.html#upsert) will upsert a record or records given in first argument. External ID field name must be specified in second argument.

```
// Single record upsert
conn.sobject("UpsertTable__c").upsert({
  Name : 'Record #1',
  ExtId__c : 'ID-0000001'
}, 'ExtId__c', function(err, ret) {
  if (err || !ret.success) { return console.error(err, ret); }
  console.log('Upserted Successfully');
  // ...
});
```

Edit Run

```
// Multiple record upsert
conn.sobject("UpsertTable__c").upsert([
 { Name : 'Record #1', ExtId__c : 'ID-0000001' },
 { Name : 'Record #2', ExtId__c : 'ID-0000002' }
],
'ExtId__c',
function(err, rets) {
  if (err) { return console.error(err); }
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("Upserted Successfully");
    }
  }
  // ...
});
```

Edit Run

Describe

Metadata description API for Salesforce object.

Describe SObject

You can use `SObject#describe()` (/jsforce/doc/SObject.html#describe) to fetch SObject metadata,

```
conn.sobject("Account").describe(function(err, meta) {
  if (err) { return console.error(err); }
  console.log('Label : ' + meta.label);
  console.log('Num of Fields : ' + meta.fields.length);
  // ...
});
```

Edit Run

or can use `Connection#describe(sobjectType)` (/jsforce/doc/Connection.html#describe) (or its synonym `Connection#describeSObject(sobjectType)` (/jsforce/doc/Connection.html#describeSObject)) alternatively.

```
conn.describe("Account", function(err, meta) {
  if (err) { return console.error(err); }
  console.log('Label : ' + meta.label);
  console.log('Num of Fields : ' + meta.fields.length);
  // ...
});
```

Edit Run

Describe Global

`SObject#describeGlobal()` (/jsforce/doc/SObject.html#describeGlobal) returns all SObject information registered in Salesforce (without detail information like fields, childRelationships).

```
conn.describeGlobal(function(err, res) {
  if (err) { return console.error(err); }
  console.log('Num of SObjects : ' + res.sobjects.length);
  // ...
});
```

Edit Run

Cached Call

Each description API has "cached" version with suffix of `$` (coming from similar pronounce "cash"), which keeps the API call result for later use.

```
// First lookup local cache, and then call remote API if cache doesn't exist.
conn.sobject("Account").describe$(function(err, meta) {
  if (err) { return console.error(err); }
  console.log('Label : ' + meta.label);
  console.log('Num of Fields : ' + meta.fields.length);
  // ...
});
```

Edit Run

```
// If you can assume it should have already cached the result,
// you can use synchronous call to access the result;
var meta = conn.sobject("Account").describe$();
console.log('Label : ' + meta.label);
console.log('Num of Fields : ' + meta.fields.length);
// ...
```

Edit Run

Cache clearance should be done explicitly by developers.

```
// Delete cache of 'Account' SObject description result
conn.sobject('Account').describe$.clear();
```

```
// Delete cache of global sobject description
conn.describeGlobal$.clear();
```

```
// Delete all API caches in connection.
conn.cache.clear();
```

Identity

`Connection#identity()` (/jsforce/doc/Connection.html#identity) is available to get current API session user identity information.

```
conn.identity(function(err, res) {
  if (err) { return console.error(err); }
  console.log("user ID: " + res.user_id);
  console.log("organization ID: " + res.organization_id);
  console.log("username: " + res.username);
  console.log("display name: " + res.display_name);
});
```

Edit Run

History

Recently Accessed Records

`SObject#recent()` (/jsforce/doc/SObject.html#recent) returns recently accessed records in the SObject.

```
conn.sobject('Account').recent(function(err, res) {
  if (err) { return console.error(err); }
  console.log(res);
});
```

Edit Run

`Connection#recent()` (/jsforce/doc/Connection.html#recent) returns records in all object types which are recently accessed.

```
conn.recent(function(err, res) {
  if (err) { return console.error(err); }
  console.log(res);
});
```

Edit Run

Recently Updated Records

`SObject#updated(startDate, endDate)` (/jsforce/doc/SObject.html#updated) returns record IDs which are recently updated.

```
conn.sobject('Account').updated('2014-02-01', '2014-02-15', function(err, res) {
  if (err) { return console.error(err); }
  console.log("Latest date covered: " + res.latestDateCovered);
  console.log("Updated records : " + res.ids.length);
});
```

Edit Run

Recently Deleted Records

`SObject#deleted(startDate, endDate)` (/jsforce/doc/SObject.html#deleted) returns record IDs which are recently deleted.

```
conn.sobject('Account').deleted('2014-02-01', '2014-02-15', function(err, res) {
  if (err) { return console.error(err); }
  console.log("Ealiest date available: " + res.earliestDateAvailable);
  console.log("Latest date covered: " + res.latestDateCovered);
  console.log("Deleted records : " + res.deletedRecords.length);
});
```

Edit Run

API Limit and Usage

`Connection#limitInfo` is a property which stores the latest API usage information.

```
console.log("API Limit: " + conn.limitInfo.apiUsage.limit);
console.log("API Used: " + conn.limitInfo.apiUsage.used);
```

Edit Run

Note that the limit information is available only after at least *one* REST API call, as it is included in response headers of API requests.

Analytics API

By using Analytics API, you can get the output result from a report registered in Salesforce.

Get Recently Used Reports

`Analytics#reports()` (/jsforce/doc/Analytics.html#reports) lists recently accessed reports.

```
// get recent reports
conn.analytics.reports(function(err, reports) {
  if (err) { return console.error(err); }
  console.log("reports length: "+reports.length);
  for (var i=0; i < reports.length; i++) {
    console.log(reports[i].id);
    console.log(reports[i].name);
  }
  // ...
});
```

Edit Run

Describe Report Metadata

`Analytics#report(reportId)` (/jsforce/doc/Analytics.html#report) gives a reference to the report object specified in `reportId`. By calling `Analytics-Report#describe()` (/jsforce/doc/Analytics-Report.html#describe), you can get the report metadata defined in Salesforce without executing the report.

You should check Analytics REST API Guide (http://www.salesforce.com/us/developer/docs/api_analytics/index.htm) to understand the structure of returned report metadata.

```
var reportId = '00O10000000pUw2EAE';
conn.analytics.report(reportId).describe(function(err, meta) {
  if (err) { return console.error(err); }
  console.log(meta.reportMetadata);
  console.log(meta.reportTypeMetadata);
  console.log(meta.reportExtendedMetadata);
});
```

Edit Run

Execute Report

Execute Synchronously

By calling `Analytics-Report#execute(options)` (/jsforce/doc/Analytics-Report.html#execute), the report is exected in Salesforce, and returns executed result synchronously. Please refer to Analytics API document about the format of retruning result.

```
// get report reference
var reportId = '00O10000000pUw2EAE';
var report = conn.analytics.report(reportId);

// execute report synchronously
report.execute(function(err, result) {
  if (err) { return console.error(err); }
  console.log(result.reportMetadata);
  console.log(result.factMap);
  console.log(result.factMap["T!T"]);
  console.log(result.factMap["T!T"].aggregates);
  // ...
});
```

Edit Run

Include Detail Rows in Execution

Setting `details` to true in `options`, it returns execution result with detail rows.

```
// execute report synchronously with details option,
// to get detail rows in execution result.
var reportId = '00O10000000pUw2EAE';
var report = conn.analytics.report(reportId);
report.execute({ details: true }, function(err, result) {
  if (err) { return console.error(err); }
  console.log(result.reportMetadata);
  console.log(result.factMap);
  console.log(result.factMap["T!T"]);
  console.log(result.factMap["T!T"].aggregates);
  console.log(result.factMap["T!T"].rows); // <= detail rows in array
  // ...
});
```

Edit Run

Override Report Metadata in Execution

You can override report behavior by putting `metadata` object in `options`. For example, following code shows how to update filtering conditions of a report on demand.

```
// overriding report metadata
var metadata = {
  reportMetadata : {
    reportFilters : [{
      column: 'COMPANY',
      operator: 'contains',
      value: ',Inc.'
    }]
  }
};
// execute report synchronously with overridden filters.
var reportId = '00O10000000pUw2EAE';
var report = conn.analytics.report(reportId);
report.execute({ metadata : metadata }, function(err, result) {
  if (err) { return console.error(err); }
  console.log(result.reportMetadata);
  console.log(result.reportMetadata.reportFilters.length); // <= 1
  console.log(result.reportMetadata.reportFilters[0].column); // <= 'COMPANY'
  console.log(result.reportMetadata.reportFilters[0].operator); // <= 'contains'
  console.log(result.reportMetadata.reportFilters[0].value); // <= ',Inc.'
  // ...
});
```

Edit Run

Execute Asynchronously

`Analytics-Report#executeAsync(options)` (/jsforce/doc/Analytics-Report.html#executeAsync) executes the report asynchronously in Salesforce, registering an instance to the report to lookup the executed result in future.

```
var instanceId;

// execute report asynchronously
var reportId = '00O10000000pUw2EAE';
var report = conn.analytics.report(reportId);
report.executeAsync({ details: true }, function(err, instance) {
  if (err) { return console.error(err); }
  console.log(instance.id); // <= registered report instance id
  instanceId = instance.id;
  // ...
});
```

Edit Run

Afterward use `Analytics-Report#instance(instanceId)` (/jsforce/doc/Analytics-Report.html#instance) and call `Analytics-ReportInstance#retrieve()` (/jsforce/doc/Analytics-ReportInstance.html#retrieve) to get the executed result.

```
// retrieve asynchronously executed result afterward.
report.instance(instanceId).retrieve(function(err, result) {
  if (err) { return console.error(err); }
  console.log(result.reportMetadata);
  console.log(result.factMap);
  console.log(result.factMap["T!T"]);
  console.log(result.factMap["T!T"].aggregates);
  console.log(result.factMap["T!T"].rows);
  // ...
});
```

Edit Run

Apex REST

If you have a static Apex class in Salesforce and are exposing it using "Apex REST" feature, you can call it by using `Apex#get(path)` (/jsforce/doc/Apex.html#get), `Apex#post(path, body)` (/jsforce/doc/Apex.html#post), `Apex#put(path, body)` (/jsforce/doc/Apex.html#put), `Apex#patch(path, body)` (/jsforce/doc/Apex.html#patch), and `Apex#del(path, body)` (/jsforce/doc/Apex.html#del) (or its synonym `Apex#delete(path, body)` (/jsforce/doc/Apex.html#delete)) through `apex` API object in connection object.

```
// body payload structure is depending to the Apex REST method interface.
var body = { title: 'hello', num : 1 };
conn.apex.post("/MyTestApexRest/", body, function(err, res) {
  if (err) { return console.error(err); }
  console.log("response: ", res);
  // the response object structure depends on the definition of apex class
});
```

Edit Run

Bulk API

JSforce package also supports Bulk API. It is not only mapping each Bulk API endpoint in low level, but also introducing utility interface in bulk load operations.

Load From Records

First, assume that you have record set in array object to insert into Salesforce.

```
//
// Records to insert in bulk.
//
var accounts = [
{ Name : 'Account #1', ... },
{ Name : 'Account #2', ... },
{ Name : 'Account #3', ... },
...
];
```

You can use `SObject#create(record)` (/jsforce/doc/SObject.html#create), but it consumes API quota per record, so not practical for large set of records. We can use bulk API interface to load them.

Similar to Salesforce Bulk API, first create bulk job by `Bulk#createJob(sobjectType, operation)` (/jsforce/doc/Bulk.html#createJob) through `bulk` API object in connection object.

Next, create a new batch in the job, by calling `Bulk-Job#createBatch()` (/jsforce/doc/Bulk-Job.html#createBatch) through the job object created previously.

```
var job = conn.bulk.createJob("Account", "insert");
var batch = job.createBatch();
```

Then bulk load the records by calling `Bulk-Batch#execute(input)` (/jsforce/doc/Bulk-Batch.html#execute) of created batch object, passing the records in `input` argument.

When the batch is queued in Salesforce, it is notified by `queue` event, and you can get job ID and batch ID.

```
batch.execute(accounts);
batch.on("queue", function(batchInfo) { // fired when batch request is queued in server.
  console.log('batchInfo:', batchInfo);
  batchId = batchInfo.id;
  jobId = batchInfo.jobId;
  // ...
});
```

After the batch is queued and job / batch ID is created, wait the batch completion by polling.

When the batch process in Salesforce has been completed, it is notified by `response` event with batch result information.

```
var job = conn.bulk.job(jobId);
var batch = job.batch(batchId);
batch.poll(1000 /* interval(ms) */, 20000 /* timeout(ms) */); // start polling
batch.on("response", function(rets) { // fired when batch is finished and result retrieved
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("#" + (i+1) + " loaded successfully, id = " + rets[i].id);
    } else {
      console.log("#" + (i+1) + " error occurred, message = " + rets[i].errors.join(', '));
    }
  }
  // ...
});
```

Below is an example of the full bulk loading flow from scratch.

```
// Provide records
var accounts = [
  { Name : 'Account #1' },
  { Name : 'Account #2' },
  { Name : 'Account #3' },
];
// Create job and batch
var job = conn.bulk.createJob("Account", "insert");
var batch = job.createBatch();
// start job
batch.execute(accounts);
// listen for events
batch.on("error", function(batchInfo) { // fired when batch request is queued in server.
  console.log('Error, batchInfo:', batchInfo);
});
batch.on("queue", function(batchInfo) { // fired when batch request is queued in server.
  console.log('queue, batchInfo:', batchInfo);
  batch.poll(1000 /* interval(ms) */, 20000 /* timeout(ms) */); // start polling - Do not poll until the batch has started
});
batch.on("response", function(rets) { // fired when batch finished and result retrieved
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("#" + (i+1) + " loaded successfully, id = " + rets[i].id);
    } else {
      console.log("#" + (i+1) + " error occurred, message = " + rets[i].errors.join(', '));
    }
  }
  // ...
});
```

Edit Run

Alternatively, you can use `Bulk#load(sobjectType, operation, input)` (/jsforce/doc/Bulk.html#load) interface to achieve the above process in one method call.

NOTE: In some cases for large data sets, a polling timeout can occur. When loading large data sets, consider changing `Bulk#pollTimeout` and `Bulk#pollInterval` property value, or using the one of the calls above with the built in `batch.poll()` or polling manually.

```
conn.bulk.pollTimeout = 25000; // Bulk timeout can be specified globally on the connection object
conn.bulk.load("Account", "insert", accounts, function(err, rets) {
  if (err) { return console.error(err); }
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("#" + (i+1) + " loaded successfully, id = " + rets[i].id);
    } else {
      console.log("#" + (i+1) + " error occurred, message = " + rets[i].errors.join(', '));
    }
  }
  // ...
});
```

Following are same calls but in different interfaces:

```
conn.sobject("Account").insertBulk(accounts, function(err, rets) {
  // ...
});
```

```
conn.sobject("Account").bulkload("insert").execute(accounts, function(err, rets) {
  // ...
});
```

To check the status of a batch job without using the built in polling methods, you can use `Bulk#check()` (/jsforce/doc/Bulk.html#check).

```
conn.bulk.job(jobId).batch(batchId).check((err, results) => {
  // Note: all returned data is of type String from parsing the XML response from Salesforce, but the following attributes are
 actually numbers: apexProcessingTime, apiActiveProcessingTime, numberRecordsFailed, numberRecordsProcessed, totalProcessingTi
me
  if (err) { return console.error(err); }
  console.log('results', results);
});
```

Load From CSV File

It also supports bulk loading from CSV file. Just use CSV file input stream as `input` argument in `Bulk#load(sobjectType, operation, input)` (/jsforce/doc/Bulk.html#load), instead of passing records in array.

```
//
// Create readable stream for CSV file to upload
//
var csvFileIn = require('fs').createReadStream("path/to/Account.csv");
//
// Call Bulk#load(sobjectType, operation, input) - use CSV file stream as "input" argument
//
conn.bulk.load("Account", "insert", csvFileIn, function(err, rets) {
  if (err) { return console.error(err); }
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("#" + (i+1) + " loaded successfully, id = " + rets[i].id);
    } else {
      console.log("#" + (i+1) + " error occurred, message = " + rets[i].errors.join(', '));
    }
  }
  // ...
});
```

Alternatively, if you have a CSV string instead of an actual file, but would still like to use the CSV data type, here is an example for node.js.

```
var s = new stream.Readable();
s.push(fileStr);
s.push(null);

var job = conn.bulk.createJob(sobject, operation, options);
var batch = job.createBatch();
batch
.execute(s)
.on("queue", function(batchInfo) {
  console.log('Apex job queued');
  // Since we useed .execute(), we need to poll until completion using batch.poll() or manually using batch.check()
  // See the previous examples for reference
})
.on("error", function(err) {
  console.log('Apex job error');
});
```

`Bulk-Batch#stream()` (/jsforce/doc/Bulk-Batch.html#stream) returns a Node.js standard writable stream which accepts batch input. You can pipe input stream to it afterward.

```
var batch = conn.bulk.load("Account", "insert");
batch.on("response", function(rets) { // fired when batch finished and result retrieved
  for (var i=0; i < rets.length; i++) {
    if (rets[i].success) {
      console.log("#" + (i+1) + " loaded successfully, id = " + rets[i].id);
    } else {
      console.log("#" + (i+1) + " error occurred, message = " + rets[i].errors.join(', '));
    }
  }
);
//
// When input stream becomes available, pipe it to batch stream.
//
csvFileIn.pipe(batch.stream());
```

Update / Delete Queried Records

If you want to update / delete records in Salesforce which match specified condition in bulk, now you don't have to write a code which download & upload records information. `Query#update(mapping)` (/jsforce/doc/Query.html#update) / `Query#destroy()` (/jsforce/doc/Query.html#destroy) will directly manipulate records.

```
// DELETE FROM Account WHERE CreatedDate = TODAY
conn.sobject('Account')
    .find({ CreatedDate : jsforce.Date.TODAY })
    .destroy(function(err, rets) {
      if (err) { return console.error(err); }
      console.log(rets);
      // ...
    });
```

Edit Run

```
// UPDATE Opportunity
// SET CloseDate = '2013-08-31'
// WHERE Account.Name = 'Salesforce.com'
conn.sobject('Opportunity')
    .find({ 'Account.Name' : 'Salesforce.com' })
    .update({ CloseDate: '2013-08-31' }, function(err, rets) {
      if (err) { return console.error(err); }
      console.log(rets);
      // ...
    });
```

Edit Run

In `Query#update(mapping)` (/jsforce/doc/Query.html#update), you can include simple templating notation in mapping record.

```
//
// UPDATE Task
// SET Description = CONCATENATE(Subject || ' ' || Status)
// WHERE ActivityDate = TODAY
//
conn.sobject('Task')
    .find({ ActivityDate : jsforce.Date.TODAY })
    .update({ Description: '${Subject}  ${Status}' }, function(err, rets) {
      if (err) { return console.error(err); }
      console.log(rets);
      // ...
    });
```

Edit Run

To achieve further complex mapping, `Query#update(mapping)` (/jsforce/doc/Query.html#update) accepts mapping function in `mapping` argument.

```
conn.sobject('Task')
    .find({ ActivityDate : jsforce.Date.TODAY })
    .update(function(rec) {
      return {
        Description: rec.Subject + ' ' + rec.Status
      }
    }, function(err, rets) {
      if (err) { return console.error(err); }
      console.log(rets);
      // ...
    });
```

Edit Run

If you are creating query object from SOQL by using `Connection#query(soql)` (/jsforce/doc/Connection.html#query), the bulk delete/update operation cannot be achieved because no sobject type information available initially. You can avoid it by passing optional argument `sobjectType` in `Query#destroy(sobjectType)` (/jsforce/doc/Query.html#destroy) or `Query#update(mapping, sobjectType)` (/jsforce/doc/Query.html#update).

```
conn.query("SELECT Id FROM Account WHERE CreatedDate = TODAY")
    .destroy('Account', function(err, rets) {
      if (err) { return console.error(err); }
      console.log(rets);
      // ...
    });
```

Edit Run

```
conn.query("SELECT Id FROM Task WHERE ActivityDate = TODAY")
    .update({ Description: '${Subject}  ${Status}' }, 'Task', function(err, rets) {
      if (err) { return console.error(err); }
      console.log(rets);
      // ...
    });
```

Edit Run

NOTE: Be careful when using this feature not to break/lose existing data in Salesforce. Careful testing is recommended before applying the code to your production environment.

Bulk Query

From ver. 1.3, additional functionality was added to the bulk query API. It fetches records in bulk in record stream, or CSV stream which can be piped out to a CSV file.

```
conn.bulk.query("SELECT Id, Name, NumberOfEmployees FROM Account")
  .on('record', function(rec) { console.log(rec); })
  .on('error', function(err) { console.error(err); });
```

Edit Run

```
var fs = require('fs');
conn.bulk.query("SELECT Id, Name, NumberOfEmployees FROM Account")
  .stream().pipe(fs.createWriteStream('./accounts.csv'));
```

If you already know the job id and batch id for the bulk query, you can get the batch result ids by calling `Batch#retrieve()` (/jsforce/doc/Batch.html#retrieve). Retrieval for each result is done by `Batch#result(resultId)` (/jsforce/doc/Batch.html#result)

```
var fs = require('fs');
var batch = conn.bulk.job(jobId).batch(batchId);
batch.retrieve(function(err, results) {
  if (err) { return console.error(err); }
  for (var i=0; i < results.length; i++) {
    var resultId = result[i].id;
    batch.result(resultId).stream().pipe(fs.createWriteStream('./result'+i+'.csv'));
  }
});
```

Chatter API

Chatter API resources can be accessed via `Chatter#resource(path)` (/jsforce/doc/Chatter.html#resource). The path for the resource can be a relative path from `/services/data/vX.X/chatter/`, `/services/data/`, or site-root relative path, otherwise absolute URI.

Please check official Chatter REST API Guide (http://www.salesforce.com/us/developer/docs/chatterapi/) to understand resource paths for chatter objects.

Get Resource Information

If you want to retrieve the information for specified resource, `Chatter-Resource#retrieve()` (/jsforce/doc/Chatter-Resource.html#retrieve) will get information of the resource.

```
conn.chatter.resource('/users/me').retrieve(function(err, res) {
  if (err) { return console.error(err); }
  console.log("username: "+ res.username);
  console.log("email: "+ res.email);
  console.log("small photo url: "+ res.photo.smallPhotoUrl);
});
```

Edit Run

Get Collection Resource Information

You can pass query parameters to collection resource, to filter result or specify offset/limit for result. All acceptable query parameters are written in Chatter REST API manual.

```
conn.chatter.resource('/users', { q: 'Suzuki' }).retrieve(function(err, result) {
  if (err) { return console.error(err); }
  console.log("current page URL: " + result.currentPageUrl);
  console.log("next page URL: " + result.nextPageUrl);
  console.log("users count: " + result.users.length);
  for (var i=0; i<result.users.length; i++) {
    var user = users[i];
    console.log('User ID: '+user.id);
    console.log('User URL: '+user.url);
    console.log('Username: '+user.username);
  }
});
```

Edit Run

Post a Feed Item

To post a feed item or a comment, use `Chatter-Resource#create(data)` (/jsforce/doc/Chatter-Resource.html#create) for collection resource.

```
conn.chatter.resource('/feed-elements').create({
  body: {
    messageSegments: [{
      type: 'Text',
      text: 'This is new post'
    }]
  },
  feedElementType : 'FeedItem',
  subjectId: 'me'
}, function(err, result) {
  if (err) { return console.error(err); }
  console.log("Id: " + result.id);
  console.log("URL: " + result.url);
  console.log("Body: " + result.body.messageSegments[0].text);
  console.log("Comments URL: " + result.capabilities.comments.page.currentPageUrl);
});
```

Edit Run

Post a Comment

You can add a comment by posting message to feed item's comments URL:

```
var commentsUrl = '/feed-elements/0D55000001j5qn8CAA/capabilities/comments/items';
conn.chatter.resource(commentsUrl).create({
  body: {
    messageSegments: [{
      type: 'Text',
      text: 'This is new comment #1'
    }]
  }
}, function(err, result) {
  if (err) { return console.error(err); }
  console.log("Id: " + result.id);
  console.log("URL: " + result.url);
  console.log("Body: " + result.body.messageSegments[0].text);
});
```

Edit Run

Add Like

You can add likes to feed items/comments by posting empty string to like URL:

```
var itemLikesUrl = '/feed-elements/0D55000001j5r2rCAA/capabilities/chatter-likes/items';
conn.chatter.resource(itemLikesUrl).create("", function(err, result) {
  if (err) { return console.error(err); }
  console.log("URL: " + result.url);
  console.log("Liked Item ID:" + result.likedItem.id);
});
```

Edit Run

Batch Operation

Using `Chatter#batch(requests)` (/jsforce/doc/Chatter.html#batch), you can execute multiple Chatter resource requests in one API call. Requests should be CRUD operations for Chatter API resource.

```
conn.chatter.batch([
  conn.chatter.resource('/feed-elements').create({
    body: {
      messageSegments: [{
        type: 'Text',
        text: 'This is a post text'
      }]
    },
    feedElementType: 'FeedItem',
    subjectId: 'me'
  }),
  conn.chatter.resource('/feed-elements').create({
    body: {
      messageSegments: [{
        type: 'Text',
        text: 'This is another post text, following to previous.'
      }]
    },
    feedElementType: 'FeedItem',
    subjectId: 'me'
  }),
  conn.chatter.resource('/feeds/news/me/feed-elements', { pageSize: 2, sort: "CreatedDateDesc" }),
], function(err, res) {
  if (err) { return console.error(err); }
  console.log("Error? " + res.hasErrors);
  var results = res.results;
  console.log("batch request executed: " + results.length);
  console.log("request #1 - status code: " + results[0].statusCode);
  console.log("request #1 - result URL: " + results[0].result.url);
  console.log("request #2 - status code: " + results[1].statusCode);
  console.log("request #2 - result URL: " + results[1].result.url);
  console.log("request #3 - status code: " + results[2].statusCode);
  console.log("request #3 - current Page URL: " + results[2].result.currentPageUrl);
});
```

Edit Run

Metadata API

Describe Metadata

`Metadata#describe(version)` (/jsforce/doc/Metadata.html#describe) is the method to list all metadata in an org.

```
conn.metadata.describe('39.0', function(err, metadata) {
  if (err) { return console.error('err', err); }
  for (var i=0; i < metadata.length; i++) {
    var meta = metadata[i];
    console.log("organizationNamespace: " + meta.organizationNamespace);
    console.log("partialSaveAllowed: " + meta.partialSaveAllowed);
    console.log("testRequired: " + meta.testRequired);
    console.log("metadataObjects count: " + metadataObjects.length);
  }
});
```

Edit Run

List Metadata

`Metadata#list(types, version)` (/jsforce/doc/Metadata.html#list) is the method to list summary information for all metadata types.

```
var types = [{type: 'CustomObject', folder: null}];
conn.metadata.list(types, '39.0', function(err, metadata) {
  if (err) { return console.error('err', err); }
    var meta = metadata[0];
    console.log('metadata count: ' + metadata.length);
    console.log('createdById: ' + meta.createdById);
    console.log('createdByName: ' + meta.createdByName);
    console.log('createdDate: ' + meta.createdDate);
    console.log('fileName: ' + meta.fileName);
    console.log('fullName: ' + meta.fullName);
    console.log('id: ' + meta.id);
    console.log('lastModifiedById: ' + meta.lastModifiedById);
    console.log('lastModifiedByName: ' + meta.lastModifiedByName);
    console.log('lastModifiedDate: ' + meta.lastModifiedDate);
    console.log('manageableState: ' + meta.manageableState);
    console.log('namespacePrefix: ' + meta.namespacePrefix);
    console.log('type: ' + meta.type);
});
```

Edit Run

Read Metadata

`Metadata#read(type, fullNames)` (/jsforce/doc/Metadata.html#read) is the method to retrieve metadata information which are specified by given names.

```
var fullNames = [ 'Account', 'Contact' ];
conn.metadata.read('CustomObject', fullNames, function(err, metadata) {
  if (err) { console.error(err); }
  for (var i=0; i < metadata.length; i++) {
    var meta = metadata[i];
    console.log("Full Name: " + meta.fullName);
    console.log("Fields count: " + meta.fields.length);
    console.log("Sharing Model: " + meta.sharingModel);
  }
});
```

Edit Run

Create Metadata

To create new metadata objects, use `Metadata#create(type, metadata)` (/jsforce/doc/Metadata.html#create). Metadata format for each metadata types are written in the Salesforce Metadata API document.

```
// creating metadata in array
var metadata = [{
  fullName: 'TestObject1__c',
  label: 'Test Object 1',
  pluralLabel: 'Test Object 1',
  nameField: {
    type: 'Text',
    label: 'Test Object Name'
  },
  deploymentStatus: 'Deployed',
  sharingModel: 'ReadWrite'
}, {
  fullName: 'TestObject2__c',
  label: 'Test Object 2',
  pluralLabel: 'Test Object 2',
  nameField: {
    type: 'AutoNumber',
    label: 'Test Object #'
  },
  deploymentStatus: 'InDevelopment',
  sharingModel: 'Private'
}];
conn.metadata.create('CustomObject', metadata, function(err, results) {
  if (err) { console.err(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('success ? : ' + result.success);
    console.log('fullName : ' + result.fullName);
  }
});
```

Edit Run

There is an alternative method to create metadata, in aynchronous - `Metadata#createAync()` (/jsforce/doc/Metadata.html#createAync).

This asynchronous version is different from synchronous one - it returns asynchronous result ids with current statuses, which can be used for later execution status query.

NOTE: This API is depricated from Salesforce as of API version 31.0 in favor of the synchronous version of the call

```
// request creating metadata and receive execution ids & statuses
var asyncResultIds = [];
conn.metadata.createAsync('CustomObject', metadata, function(err, results) {
  if (err) { console.err(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('id: ' + result.id);
    console.log('done ? : ' + result.done);
    console.log('state : ' + result.state); console.log(results);
    // save for later status check
    asyncResultIds.push(result.id);
  }
});
```

And then you can check creation statuses by `Metadata#checkStatus(asyncResultIds)` (/jsforce/doc/Metadata.html#checkStatus), and wait their completion by calling `Metadata-AsyncResultLocator#complete()` (/jsforce/doc/Metadata-AsyncResultLocator.html#complete) for returned object.

```
conn.metadata.checkStatus(asyncResultIds).complete(function(err, results) {
  if (err) { console.error(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('id: ' + result.id);
    console.log('done ? : ' + result.done);
    console.log('state : ' + result.state);
  }
});
```

Or you can directly apply `Metadata-AsyncResultLocator#complete()` (/jsforce/doc/Metadata-AsyncResultLocator.html#complete) call for the locator object returned from `Metadata#createAsync()` (/jsforce/doc/Metadata.html#createAsync) call.

```
conn.metadata.createAsync('CustomObject', metadata).complete(function(err, results) {
  if (err) { console.err(err); }
  console.log(results);
});
`
```

NOTE: In version 1.2.x, `Metadata#create()` (/jsforce/doc/Metadata.html#create) method was an alias of `Metadata#createAsync()` (/jsforce/doc/Metadata.html#createAsync).

From ver 1.3, the method has been changed to point to synchronous call `Metadata#createSync()` (/jsforce/doc/Metadata.html#createSync) which is corresponding to the sync API newly introduced from API 30.0. This is due to the removal of asynchronous metadata call from API 31.0.

Asynchronous method `Metadata#createAsync()` (/jsforce/doc/Metadata.html#createAsync) still works if API version is specified to less than 31.0, but not recommended for active usage.

Update Metadata

`Metadata#update(type, updateMetadata)` (/jsforce/doc/Metadata.html#update) can be used for updating existing metadata objects.

```
/* @interactive */
var metadata = [{
  fullName: 'TestObject1__c.AutoNumberField__c',
  label: 'Auto Number #2',
  length: 50
}]
conn.metadata.update('CustomField', metadata, function(err, results) {
  if (err) { console.error(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('success ? : ' + result.success);
    console.log('fullName : ' + result.fullName);
  }
});
```

NOTE: In version 1.2.x, `Metadata#update()` (/jsforce/doc/Metadata.html#update) method was an alias of `Metadata#updateAsync()` (/jsforce/doc/Metadata.html#updateAsync).

From ver 1.3, the method has been changed to point to synchronous call `Metadata#updateSync()` (/jsforce/doc/Metadata.html#updateSync) which is corresponding to the sync API newly introduced from API 30.0. This is due to the removal of asynchronous metadata call from API 31.0.

Asynchronous method `Metadata#updateAsync()` (/jsforce/doc/Metadata.html#updateAsync) still works if API version is specified to less than 31.0, but not recommended for active usage.

Upsert Metadata

`Metadata#upsert(type, metadata)` (/jsforce/doc/Metadata.html#upsert) is used for upserting metadata - insert new metadata when it is not available, otherwise update it.

```
var metadata = [{
  fullName: 'TestObject2__c',
  label: 'Upserted Object 2',
  pluralLabel: 'Upserted Object 2',
  nameField: {
    type: 'Text',
    label: 'Test Object Name'
  },
  deploymentStatus: 'Deployed',
  sharingModel: 'ReadWrite'
}, {
  fullName: 'TestObject__c',
  label: 'Upserted Object 3',
  pluralLabel: 'Upserted Object 3',
  nameField: {
    type: 'Text',
    label: 'Test Object Name'
  },
  deploymentStatus: 'Deployed',
  sharingModel: 'ReadWrite'
}];
conn.metadata.upsert('CustomObject', metadata, function(err, results) {
  if (err) { console.error(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('success ? : ' + result.success);
    console.log('created ? : ' + result.created);
    console.log('fullName : ' + result.fullName);
  }
});
```

Edit Run

Rename Metadata

`Metadata#rename(type, oldFullName, newFullName)` (/jsforce/doc/Metadata.html#rename) is used for renaming metadata.

```
conn.metadata.rename('CustomObject', 'TestObject3__c', 'UpdatedTestObject3__c', function(err, result) {
  if (err) { console.error(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('success ? : ' + result.success);
    console.log('fullName : ' + result.fullName);
  }
});
```

Edit Run

Delete Metadata

`Metadata#delete(type, metadata)` (/jsforce/doc/Metadata.html#delete) can be used for deleting existing metadata objects.

```
var fullNames = ['TestObject1__c', 'TestObject2__c'];
conn.metadata.delete('CustomObject', fullNames, function(err, results) {
  if (err) { console.error(err); }
  for (var i=0; i < results.length; i++) {
    var result = results[i];
    console.log('success ? : ' + result.success);
    console.log('fullName : ' + result.fullName);
  }
});
```

Edit Run

NOTE: In version 1.2.x, `Metadata#delete()` (/jsforce/doc/Metadata.html#delete) method was an alias of `Metadata#deleteAsync()` (/jsforce/doc/Metadata.html#deleteAsync).

From ver 1.3, the method has been changed to point to synchronous call `Metadata#deleteSync()` (/jsforce/doc/Metadata.html#deleteSync) which is corresponding to the sync API newly introduced from API 30.0. This is due to the removal of asynchronous metadata call from API 31.0.

Asynchronous method `Metadata#deleteAsync()` (/jsforce/doc/Metadata.html#deleteAsync) still works if API version is specified to less than 31.0, but not recommended for active usage.

Retrieve / Deploy Metadata (File-based)

You can retrieve metadata information which is currently registered in Salesforce, `Metadata#retrieve(options)` (/jsforce/doc/Metadata.html#retrieve) command can be used.

The structure of hash object argument `options` is same to the message object defined in Salesforce Metadata API.

```
var fs = require('fs');
conn.metadata.retrieve({ packageNames: [ 'My Test Package' ] })
            .stream().pipe(fs.createWriteStream("./path/to/MyPackage.zip"));
```

If you have metadata definition files in your file system, create zip file from them and call `Metadata#deploy(zipIn, options)` (/jsforce/doc/Metadata.html#deploy) to deploy all of them.

```
var fs = require('fs');
var zipStream = fs.createReadStream("./path/to/MyPackage.zip");
conn.metadata.deploy(zipStream, { runTests: [ 'MyApexTriggerTest' ] })
  .complete(function(err, result) {
    if (err) { console.error(err); }
    console.log('done ? :' + result.done);
    console.log('success ? : ' + result.true);
    console.log('state : ' + result.state);
    console.log('component errors: ' + result.numberComponentErrors);
    console.log('components deployed: ' + result.numberComponentsDeployed);
    console.log('tests completed: ' + result.numberTestsCompleted);
  });
```

Streaming API

You can subscribe topic and receive message from Salesforce Streaming API, by using `Streaming#Topic(topicName)` (/jsforce/doc/Streaming.html#Topic) and `Streaming-Topic#subscribe(listener)` (/jsforce/doc/Streaming-Topic.html#subscribe).

Before the subscription, you should insert appropriate PushTopic record (in this example, "InvoiceStatementUpdates") as written in Streaming API guide (http://www.salesforce.com/us/developer/docs/api_streaming/).

```
conn.streaming.topic("InvoiceStatementUpdates").subscribe(function(message) {
  console.log('Event Type : ' + message.event.type);
  console.log('Event Created : ' + message.event.createdDate);
  console.log('Object Id : ' + message.sobject.Id);
});
```

Tooling API

Tooling API is used to build custom development tools for Salesforce platform, for example building custom Apex Code / Visualforce page editor.

Tooling API has almost same interface as usual REST API, so CRUD operation, query, and describe can be done also for these developer objects.

CRUD to Tooling Objects

You can create/retrieve/update/delete records in tooling objects (e.g. ApexCode, ApexPage).

To get reference of tooling object, use `Tooling#sobject(sobjectType)` (/jsforce/doc/Tooling.html#sobject).

```
var apexBody = [
  "public class TestApex {",
  "  public string sayHello() {",
  "    return 'Hello';",
  "  }",
  "}"
].join('\n');
conn.tooling.sobject('ApexClass').create({
  body: apexBody
}, function(err, res) {
  if (err) { return console.error(err); }
  console.log(res);
});
```

Edit Run

Query Tooling Objects

Querying records in tooling objects is also supported. Use `Tooling#query(soql)` (/jsforce/doc/Tooling.html#query) or `SObject#find(filters, fields)` (/jsforce/doc/SObject.html#find).

```
conn.tooling.sobject('ApexTrigger')
  .find({ TableEnumOrId: "Lead" })
  .execute(function(err, records) {
    if (err) { return console.error(err); }
    console.log("fetched : " + records.length);
    for (var i=0; i < records.length; i++) {
      var record = records[i];
      console.log('Id: ' + record.Id);
      console.log('Name: ' + record.Name);
    }
  });
```

Edit Run

Describe Tooling Objects

Describing all tooling objects in the organization is done by calling `Tooling#describeGlobal()` (/jsforce/doc/Tooling.html#describeGlobal).

```
conn.tooling.describeGlobal(function(err, res) {
  if (err) { return console.error(err); }
  console.log('Num of tooling objects : ' + res.sobjects.length);
  // ...
});
```

Edit Run

Describing each object detail is done by calling `SObject#describe()` (/jsforce/doc/SObject.html#describe) to tooling object reference, or just calling `Tooling#describeSObject(sobjectType)` (/jsforce/doc/Tooling.html#describeSObject).

```
conn.tooling.sobject('ApexPage').describe(function(err, meta) {
  if (err) { return console.error(err); }
  console.log('Label : ' + meta.label);
  console.log('Num of Fields : ' + meta.fields.length);
  // ...
});
```

Edit Run

Execute Anonymous Apex

You can use Tooling API to execute anonymous Apex Code, by passing apex code string text to `Tooling#executeAnonymous` .

```
// execute anonymous Apex Code
var apexBody = "System.debug('Hello, World');";
conn.tooling.executeAnonymous(apexBody, function(err, res) {
  if (err) { return console.error(err); }
  console.log("compiled?: " + res.compiled); // compiled successfully
  console.log("executed?: " + res.success); // executed successfully
  // ...
});
```

Edit Run

Advanced Topics

Record Stream Pipeline

Record stream is a stream system which regards records in its stream, similar to Node.js's standard readable/writable streams.

Query object - usually returned by `Connection#query(soql)` (/jsforce/doc/Connection.html#query) / `SObject#find(conditions, fields)` (/jsforce/doc/SObject.html#find) methods - is considered as `InputRecordStream` which emits event `record` when received record from server.

Batch object - usually returned by `Bulk-Job#createBatch()` (/jsforce/doc/Bulk-Job.html#createBatch) / `Bulk#load(sobjectType, operation, input)` (/jsforce/doc/Bulk.html#load) / `SObject#bulkload(operation, input)` (/jsforce/doc/SObject.html#bulkload) methods - is considered as `OutputRecordStream` and have `send()` and `end()` method to accept incoming record.

You can use `InputRecordStream#pipe(outputRecordStream)` (/jsforce/doc/InputRecordStream.html#pipe) to pipe record stream.

RecordStream can be converted to usual Node.js's stream object by calling `RecordStream#stream()` (/jsforce/doc/RecordStream.html#stream) method.

By default (and only currently) records are serialized to CSV string.

Piping Query Record Stream to Batch Record Stream

The idea of record stream pipeline is the base of bulk operation for queried record. For example, the same process of `Query#destroy()` (/jsforce/doc/Query.html#destroy) can be expressed as following:

```
//
// This is much more complex version of Query#destroy().
//
var Account = conn.sobject('Account');
Account.find({ CreatedDate: { $lt: jsforce.Date.LAST_YEAR }})
       .pipe(Account.deleteBulk())
       .on('response', function(rets){
         // ...
       })
       .on('error', function(err) {
         // ...
       });
```

And `Query#update(mapping)` (/jsforce/doc/Query.html#update) can be expressed as following:

```
//
// This is much more complex version of Query#update().
//
var Opp = conn.sobject('Opportunity');
Opp.find({ "Account.Id" : accId },
         { Id: 1, Name: 1, "Account.Name": 1 })
   .pipe(jsforce.RecordStream.map(function(r) {
     return { Id: r.Id,
              Name: r.Account.Name + ' - ' + r.Name };
   }))
   .pipe(Opp.updateBulk())
   .on('response', function(rets) {
     // ...
   })
   .on('error', function(err) {
     // ...
   });
```

Following is an example using `Query#stream()` (/jsforce/doc/Query.html#stream) (inherited `RecordStream#stream()` (/jsforce/doc/RecordStream.html#stream)) to convert record stream to Node.js stream, in order to export all queried records to CSV file.

```
var csvFileOut = require('fs').createWriteStream('path/to/Account.csv');
conn.query("SELECT Id, Name, Type, BillingState, BillingCity, BillingStreet FROM Account")
    .stream() // Convert to Node.js's usual readable stream.
    .pipe(csvFileOut);
```

Record Stream Filtering / Mapping

You can also filter / map queried records to output record stream. Static functions like `InputRecordStream#map(mappingFn)` (/jsforce/doc/InputRecordStream.html#map) and `InputRecordStream#filter(filterFn)` (/jsforce/doc/InputRecordStream.html#filter) create a record stream which accepts records from upstream and pass to downstream, applying given filtering / mapping function.

```
//
// Write down Contact records to CSV, with header name converted.
//
conn.sobject('Contact')
    .find({}, { Id: 1, Name: 1 })
    .map(function(r) {
      return { ID: r.Id, FULL_NAME: r.Name };
    })
    .stream().pipe(fs.createWriteStream("Contact.csv"));
//
// Write down Lead records to CSV file,
// eliminating duplicated entry with same email address.
//
var emails = {};
conn.sobject('Lead')
    .find({}, { Id: 1, Name: 1, Company: 1, Email: 1 })
    .filter(function(r) {
      var dup = emails[r.Email];
      if (!dup) { emails[r.Email] = true; }
      return !dup;
    })
    .stream().pipe(fs.createWriteStream("Lead.csv"));
```

Here is much lower level code to achieve the same result using `InputRecordStream#pipe()` (/jsforce/doc/InputRecordStream.html#pipe).

```
//
// Write down Contact records to CSV, with header name converted.
//
conn.sobject('Contact')
    .find({}, { Id: 1, Name: 1 })
    .pipe(jsforce.RecordStream.map(function(r) {
      return { ID: r.Id, FULL_NAME: r.Name };
    }))
    .stream().pipe(fs.createWriteStream("Contact.csv"));
//
// Write down Lead records to CSV file,
// eliminating duplicated entry with same email address.
//
var emails = {};
conn.sobject('Lead')
    .find({}, { Id: 1, Name: 1, Company: 1, Email: 1 })
    .pipe(jsforce.RecordStream.filter(function(r) {
      var dup = emails[r.Email];
      if (!dup) { emails[r.Email] = true; }
      return !dup;
    }))
    .stream().pipe(fs.createWriteStream("Lead.csv"));
```

Example: Data Migration

By using record stream pipeline, you can achieve data migration in a simple code.

```
//
// Connection for org which migrating data from
//
var conn1 = new jsforce.Connection({
  // ...
});
//
// Connection for org which migrating data to
//
var conn2 = new jsforce.Connection({
  // ...
});
//
// Get query record stream from Connetin #1
// and pipe it to batch record stream from connection #2
//
var query = conn1.query("SELECT Id, Name, Type, BillingState, BillingCity, BillingStreet FROM Account");
var job = conn2.bulk.createJob("Account", "insert");
var batch = job.createBatch();
query.pipe(batch);
batch.on('queue', function() {
  jobId = job.id;
  batchId = batch.id;
  //...
})
```

Home (/)