SOQL Builder

Description

A developer friendly tool for creating SOQL queries as strings. Use whenever you use Database.query() as opposed to [select …].

Code that builds a SOQL query via string concatenation is error prone. Many times you'll forget a comma, misplace an AND, mis-format a date string, or worst of all, forgot to escape user input. These types of mistakes are silly because they happen to everyone and they seem to happen over and over again. SOQL Builder's value lay in mitigating these mistakes.

Main class is **dz.**SoqlBuilder.

Apex Classes

Basic Example

Given this SOQL query:

```
SELECT name FROM account WHERE employees < 10
```

Here is the corresponding code to create the query string via a SoqlBuilder:

```
new dz.SoqlBuilder()
  .selectx('name')
  .fromx('account')
  .wherex(new dz.FieldCondition('employees').lessThan(10))
  .toSoql();
```

NOTE: You probably noticed the x's at the end of some of the methods. Unfortunately, "select", "from", and "where" are all reserved words in apex and cannot be used as method identifiers. In order to get around this, x has been appended to the end of the method name (this convention is followed throughout other classes as well). Another thing you might have noticed is the methods are "chained". That helps make this code less verbose and feel more like a SOQL statement.

Benefits

Here's a list of benefits to building SOQL queries using SoqlBuilder:

1. Reduces the risk of a silly SOQL grammar error
2. More readable / less verbose code
3. Automatic literal conversion
4. String escaping by default
5. Easy wild-carding

The first benefit is the reduced risk of misplacing an element. All String concatenation with **dz.**SoqlBuilder takes place inside the super intelligent toSoql() method.

While it might not be true for simple cases (like the example above), using a **dz.**SoqlBuilder will make your code less verbose and as a result, more readable. The construction of most real-world soql queries is an ugly, un-readable mess of String concatenation. Perhaps you've written or seen code like the following:

```
1  final Datetime aDatetime = DateTime.newInstance(2010,1,1,1,1,1);
2  final String aName = 'O\'Neal';
3  final List<String> aList = new List<String>{'Apparel','Auto'};
4  String soql = 'SELECT id,name,ispartner,industry';
5  soql += ' FROM account';
6  soql += ' WHERE CreatedDate < ';
7  soql += aDatetime.format('yyyy-MM-dd') + 'T' + aDatetime.format('hh:mm:ss') + 'Z';
8  soql += ' AND Name like \'%';
9  soql += String.escapeSingleQuotes(aName);
10 soql += '%\' AND industry INCLUDES (';
11 Boolean isFirst = true;
12 for(String anItem : aList){
13  if(isFirst){
14  isFirst = false;
15  } else {
16  soql += ',';
17  }
18  soql += '\'';
19  soql += anItem;
```

```
20    soql += '\'';
22    soql += ')';
23    System.debug(soql);
```

There's no other way to put it: dude, that's ugly code!

Here's how the same query can be constructed using SoqlBuilder:

```
1   final Datetime aDatetime = DateTime.newInstance(2010,1,1,1,1,1);
2   final String aName = 'acme';
3   final List<String> aList = new List<String>{'Apparel','Auto'};
4   String soql = new dz.SoqlBuilder()
5      .selectx(new Set<Object>{'id','name','ispartner','industry'})
6      .fromx('account')
7      .wherex(new dz.AndCondition()
8         .add(new dz.FieldCondition('CreatedDate').lessThan(aDatetime))
9         .add(new dz.FieldCondition('Name').likex(aName))
10     .add(new dz.SetCondition('industry').includes(aList))
11     )
12     .toSoql(new dz.SoqlOptions().wildcardStringsInLikeOperators());
13  System.debug(soql);
```

The third benefit is automatic conversion of literals. From the example above (the **dz.**SoqlBuilder portion), notice how the aDatetime variable is simply passed to the lessThan() method? The **dz.**FieldCondition handles converting the date time to the appropriate format. *Just in case you're curious where that code is, see the toLiteral() method in the* **dz.***SoqlUtils class.*

Another benefit which can be seen in the previous example is automatic escaping of single quotes. Notice that the aName variable is simply passed to the likex() method. When toSoql() is executed, all single quotes in all Strings will be automatically escaped! Imagine how much the AppExchange Security Review Team will like that!

Also, the previous example also shows how easily all Strings in LIKE operators can be wild-carded. By default, the "wildcardStringsInLikeOperators" property is set to FALSE. However, if you call the wildcardStringsInLikeOperators() method on a new **dz.**SoqlOptions object, then it will do just that: all strings will be wild-carded on both sides.

Reference

For all examples below, the **new dz.SoqlBuilder()** and **.toSoql()** portions are omitted. If you'd like to run one of the examples - using anonymous execute for example - then use the following snippet:

```
System.debug('\n\n'
+ new dz.SoqlBuilder()
//insert example from below here
.toSoql()
+ '\n\n');
```

SELECT

Selecting Fields

```
.selectx('ID')
.selectx('Name')
.fromx('Account')
//-> SELECT Name,ID FROM Account
```

```
.selectx(new Set<String>{'ID','Name'})
 .fromx('Account')
 //-> SELECT Name,ID FROM Account
```

```
.selectx(new List<String>{'ID','Name'})
.fromx('Account')
//-> SELECT Name,ID FROM Account
```

```
.fromx('Account')
//-> SELECT ID FROM Account
```

count()

```
.selectCount()
.fromx('Account')
//-> SELECT count() FROM Account
```

toLabel

```
.selectx(new dz.Field('Rating').toLabelx())
.fromx('Account')
//-> SELECT toLabel(Rating) FROM Account
```

Relationship Queries

```
.selectx('id')
```

```
.selectx(
   new dz.SoqlBuilder()
   .selectx('id')
   .fromx('OpportunityLineItems'))
.fromx('Opportunity')
//-> SELECT id,(SELECT id FROM OpportunityLineItems) FROM Opportunity
```

FROM

```
.fromx('account')
//-> SELECT id FROM account
```

```
.fromx('Contact c, c.Account a')
//-> SELECT id FROM Contact c, c.Account a
```

WHERE

Field Condition

```
/*
You can create a field condition using any of the following formats:
new dz.FieldCondition().field(fieldName).operator(value)
new dz.FieldCondition(fieldName).operator(value)
new dz.FieldCondition(fieldName,Operator,value)
*/
//the following four examples are equivalent:
.fromx('account').wherex(new dz.FieldCondition().field('name').equals('acme'))
.fromx('account').wherex(new dz.FieldCondition('name').equals('acme'))
.fromx('account').wherex(new dz.FieldCondition('name', dz.Operator.EQUALS,'acme'))
.fromx('account').wherex(new dz.FieldCondition('name','acme'))  //special case only valid for equals
```

```
//-> SELECT id FROM account WHERE name = 'acme'
```

Field Operators (using Operator as constructor argument)

```
/*
+-------------------------+----------+
| enum value              | operator |
+-------------------------+ ---------- +
| EQUALS                  | =        |
| NOT_EQUALS              | !=       |
| LESS_THAN               | <        |
| LESS_THAN_OR_EQUAL_TO   | <=       |
| GREATER_THAN            | >        |
| GREATER_THAN_OR_EQUAL_TO| >=       |
| LIKEX                   | like     |
+-------------------------+----------+
*/
.fromx('account').wherex(new dz.FieldCondition('employees', dz.Operator.EQUALS,1))
//-> SELECT id FROM account WHERE employees =  1
.fromx('account').wherex(new dz.FieldCondition('employees', dz.Operator.NOT_EQUALS,1))
//-> SELECT id FROM account WHERE employees !=  1
.fromx('account').wherex(new dz.FieldCondition('employees', dz.Operator.LESS_THAN,1))
//-> SELECT id FROM account WHERE employees <  1
.fromx('account').wherex(new dz.FieldCondition('employees', dz.Operator.LESS_THAN_OR_EQUAL_TO,1))
//-> SELECT id FROM account WHERE employees <=  1
.fromx('account').wherex(new dz.FieldCondition('employees', dz.Operator.GREATER_THAN,1))
//-> SELECT id FROM account WHERE employees >  1
.fromx('account').wherex(new dz.FieldCondition('employees', dz.Operator.GREATER_THAN_OR_EQUAL_TO,1))
//-> SELECT id FROM account WHERE employees >=  1
.fromx('account').wherex(new dz.FieldCondition('name', dz.Operator.LIKEX,'acme'))
//-> SELECT id FROM account WHERE name like  'acme'
```

Field Operators (operator as method identifier)

```
/*
+---------------------+----------+
| method identifier   | operator |
+---------------------+----------+
| equals              | =        |
| notEquals           | !=       |
| lessThan            | <        |
| lessThanOrEqualTo   | <=       |
```

```
| greaterThan             | >       |
| greaterThanOrEqualTo    | >=      |
| likex                   | like    |
+-------------------------+---------+
*/
.fromx('account').wherex(new dz.FieldCondition('employees').equals(1))
//-> SELECT id FROM account WHERE employees =  1
.fromx('account').wherex(new dz.FieldCondition('employees').notEquals(1))
//-> SELECT id FROM account WHERE employees !=  1
.fromx('account').wherex(new dz.FieldCondition('employees').lessThan(1))
//-> SELECT id FROM account WHERE employees <  1
.fromx('account').wherex(new dz.FieldCondition('employees').lessThanOrEqualTo(1))
//-> SELECT id FROM account WHERE employees <=  1
.fromx('account').wherex(new dz.FieldCondition('employees'.greaterThan(1))
//-> SELECT id FROM account WHERE employees >  1
.fromx('account').wherex(new dz.FieldCondition('employees').greaterThanOrEqualTo(1))
//-> SELECT id FROM account WHERE employees >=  1
.fromx('account').wherex(new dz.FieldCondition('name').likex('acme'))
//-> SELECT id FROM account WHERE name like  'acme'
```

Set Conditions

```
/*
You can create a set condition using any of the following formats:
new SetCondition().field(fieldName).operator(values)
new SetCondition(fieldName).operator(values)
new SetCondition(fieldName,Operator,values)
*/
//the following three examples are equivalent:
.fromx('account').wherex(new dz.SetCondition().field('x').includes(new List<Object>{1,2}))
.fromx('account').wherex(new dz.SetCondition('x').includes(new List<Object>{1,2}))
.fromx('account').wherex(new dz.SetCondition('x', dz.Operator.INCLUDES,new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x INCLUDES (1,2)
```

Set Operators (using Operator as constructor argument)

```
/*
+------------+----------+
| enum value | operator |
+------------+----------+
| INCLUDES   | includes |
| EXCLUDES   | excludes |
```

```
| INX          | in         |
| NOT_IN       | not in     |
+--------------+------------+
*/
.fromx('account').wherex(new dz.SetCondition('x', dz.Operator.INCLUDES,new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x INCLUDES (1,2)
.fromx('account').wherex(new dz.SetCondition('x', dz.Operator.EXCLUDES,new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x EXCLUDES (1,2)
.fromx('account').wherex(new dz.SetCondition('x', dz.Operator.INX,new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x IN (1,2)
.fromx('account').wherex(new dz.SetCondition('x', dz.Operator.NOT_IN,new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x NOT IN (1,2)
```

Set Operators (operator as method identifier)

```
/*
+--------------------+----------+
| method identifier  |operator  |
+--------------------+----------+
| includes           | includes |
| excludes           | excludes |
| inx                | in       |
| notin              | not in   |
+--------------------+----------+
*/
.fromx('account').wherex(new dz.SetCondition('x').includes(new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x INCLUDES (1,2)
.fromx('account').wherex(new dz.SetCondition('x').excludes(new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x EXCLUDES (1,2)
.fromx('account').wherex(new dz.SetCondition('x').inx(new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x IN (1,2)
.fromx('account').wherex(new dz.SetCondition('x').notIn(new List<Object>{1,2}))
//-> SELECT id FROM account WHERE x NOT IN (1,2)
```

Primitives to String literals

```
//null
.fromx('account').wherex(new dz.FieldCondition('x').equals(null))
//->SELECT id FROM account WHERE x = null
//Boolean
.fromx('account').wherex(new dz.FieldCondition('x').equals(true))
//->SELECT id FROM account WHERE x = true
```

```
//String
.fromx('account').wherex(new dz.FieldCondition('x').equals('acme'))
//->SELECT id FROM account WHERE x = 'acme'
//Integer
.fromx('account').wherex(new dz.FieldCondition('x').equals(1))
//->SELECT id FROM account WHERE x = 1
//Long
.fromx('account').wherex(new dz.FieldCondition('x').equals(1L))
//->SELECT id FROM account WHERE x = 1
//Double
.fromx('account').wherex(new dz.FieldCondition('x').equals(1.1))
//->SELECT id FROM account WHERE x = 1.1
//Date
.fromx('account').wherex(new dz.FieldCondition('x').equals(Date.newinstance(2010,1,1)))
//->SELECT id FROM account WHERE x = 2010-01-01
//Datetime
.fromx('account').wherex(new dz.FieldCondition('x').equals(Datetime.newinstance(2010,1,1,1,1,1)))
//->SELECT id FROM account WHERE x = 2010-01-01T01:01:01Z
```

Date Formulas

```
//=========================
// Hard-coded day methods
//=========================
.fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().todayx()))
//->SELECT id FROM account WHERE CreatedDate = TODAY
.fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().yesterdayx()))
//->SELECT id FROM account WHERE CreatedDate = YESTERDAY
.fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().tomorrowx()))
//->SELECT id FROM account WHERE CreatedDate = TOMORROW
.fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last90Days()))
//->SELECT id FROM account WHERE CreatedDate = LAST_90_DAYS
.fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().next90Days()))
//->SELECT id FROM account WHERE CreatedDate = NEXT_90_DAYS
//=========================
// By Units
//=========================
/*
+----------------------+-----------------+
| UnitOfTime enum value | SOQL equivalent |
+----------------------+-----------------+
| Day                  | DAY             |
| Week                 | WEEK            |
```

```
23    | Month                | MONTH          |
24    | Quarter              | QUARTER        |
25    | Year                 | FISCAL_QUARTER |
26    | FiscalQuarter        | YEAR           |
27    | FiscalYear           | FISCAL_YEAR    |
28    +----------------------+----------------+
28    */
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.Day)))
29    //->SELECT id FROM account WHERE CreatedDate = LAST_N_DAYS:1
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.Week)))
30    //->SELECT id FROM account WHERE CreatedDate = LAST_WEEK
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.Month)))
31    //->SELECT id FROM account WHERE CreatedDate = LAST_MONTH
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.Quarter)
32    //->SELECT id FROM account WHERE CreatedDate = LAST_QUARTER
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.Year)))
33    //->SELECT id FROM account WHERE CreatedDate = LAST_YEAR
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.FiscalQu
34    //->SELECT id FROM account WHERE CreatedDate = LAST_FISCAL_QUARTER
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.FiscalYe
35    //->SELECT id FROM account WHERE CreatedDate = LAST_FISCAL_YEAR
44    //========================
45    // By Interval
46    //========================
.fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().next(UnitOfTime.Day)))
47    //->SELECT id FROM account WHERE CreatedDate = NEXT_N_DAYS:1
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(UnitOfTime.Day)))
48    //->SELECT id FROM account WHERE CreatedDate = LAST_N_DAYS:1
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().next(7,UnitOfTime.Day)))
49    //->SELECT id FROM account WHERE CreatedDate = NEXT_N_DAYS:7
      .fromx('account').wherex(new dz.FieldCondition('CreatedDate',new dz.DateFormula().last(7,UnitOfTime.Day)))
50    //->SELECT id FROM account WHERE CreatedDate = LAST_N_DAYS:7
```

AND, OR & NOT

```
1   //simple AND condition
2   .fromx('account')
3   .wherex(
4     new dz.AndCondition()
```

```
5         .add(new dz.FieldCondition('name','acme'))
  .add(new dz.FieldCondition('ispartner',true)) 7)
8    //->SELECT id FROM account WHERE (name = 'acme' AND ispartner =  true)
```

```
1    //simple OR condition
2    .fromx('account')
3    .wherex(
4       new dz.OrCondition()
5         .add(new dz.FieldCondition('name','acme'))
       .add(new dz.FieldCondition('ispartner',true))
7         )
8    //->SELECT id FROM account WHERE (name = 'acme' OR ispartner =  true)//simple NOT condition
1    .fromx('account')
2    .wherex(
3       new dz.NotCondition(new AndCondition()
4         .add(new dz.FieldCondition('name','acme'))
      .add(new dz.FieldCondition('ispartner',true)))
7         )
8    //->SELECT id FROM account WHERE NOT((name = 'acme' AND ispartner = true))
```

```
1    //nested ANDs and ORs
2    .fromx('account')
3    .wherex(
4          new dz.NotCondition(
5                new dz.AndCondition()
6                .add(
7                      new dz.OrCondition()
8                      .add(new dz.FieldCondition('name','acme'))
9                      .add(
10                           new dz.AndCondition()
11                           .add(new dz.FieldCondition('ispartner',true))
12                           .add(new dz.FieldCondition('NumberOfEmployees').lessThan(10))
13                     )
14               )
15               .add(
16                     new dz.OrCondition()
17                .add(new dz.FieldCondition('createddate').lessThan(new dz.DateFormula().yesterdayx()))
18                     .add(new dz.FieldCondition('Rating','Hot'))
19               )
20         )
21   )
22   //->SELECT id FROM account WHERE NOT(((name = 'acme' OR (ispartner = true AND NumberOfEmployees        <  10
```

ORDER BY

Single Order By

```
/*
You can create an OrderBy using the following formats:
new OrderBy(fieldName).[ascending|descending|nullsFirst|nullsLast]*()
*/
.fromx('account').orderByx(new dz.OrderBy('name'))
//->SELECT id FROM account ORDER BY name
.fromx('account').orderByx(new dz.OrderBy('name').ascending().nullsFirst())
//->SELECT id FROM account ORDER BY name ASC NULLS FIRST
.fromx('account').orderByx(new dz.OrderBy('name').ascending().nullsLast())
//->SELECT id FROM account ORDER BY name ASC NULLS FIRST
.fromx('account').orderByx(new dz.OrderBy('name').descending().nullsFirst())
//->SELECT id FROM account ORDER BY name DESC NULLS FIRST
.fromx('account').orderByx(new dz.OrderBy('name').descending().nullsLast())
//->SELECT id FROM account ORDER BY name DESC NULLS FIRST
```

Multiple Order By

```
.fromx('account').orderByx(new List<OrderBy>{
   new dz.OrderBy('name').ascending().nullsFirst()
,new dz.OrderBy('rating').descending().nullsLast()
})
//->SELECT id FROM account ORDER BY name ASC NULLS FIRST, rating DESC NULLS LAST
```

LIMIT

```
.fromx('account').limitx(50)
//->SELECT id FROM account LIMIT 50
```

SOQL Options for toSoql() method

Wildcards - Enabled by default:No

```
.fromx('account')
```

```
.wherex(
   new dz.OrCondition()
   .add(new   dz.FieldCondition('name').likex('acme'))
   .add(new   dz.FieldCondition('name').likex('test'))
).toSoql()
//->SELECT id FROM account WHERE (name like 'acme' OR name like 'test')
```

```
.fromx('account')
.wherex(
   new dz.OrCondition()
   .add(new   dz.FieldCondition('name').likex('acme'))
   .add(new   dz.FieldCondition('name').likex('test'))
).toSoql(new dz.SoqlOptions().wildcardStringsInLikeOperators())
//->SELECT id FROM account WHERE (name like '%acme%' OR name like '%test%')
```

```
.fromx('account')
.wherex(
   new dz.OrCondition()
   .add(new   dz.FieldCondition('name').likex('acme'))
   .add(new   dz.FieldCondition('name').likex('test'))
).toSoql(new dz.SoqlOptions().doNotWildcardStringsInLikeOperators())
//->SELECT id FROM account WHERE (name like 'acme' OR name like 'test')
```

String Escaping

Enabled by default: Yes

```
//GOOD  (default)
.fromx('account')
.wherex(new dz.FieldCondition('name').likex('O\'Neal'))
.toSoql()
//->SELECT id FROM account WHERE name like 'O\'Neal'
```

```
//BAD!   The generated query below is invalid and will throw an error.
//Why even allow it as an option?  Because you never know when it might be needed - invalid or not.
.fromx('account')
.wherex(new dz.FieldCondition('name').likex('O\'Neal'))
.toSoql(new   dz.SoqlOptions().doNotEscapeSingleQuotes())
//->SELECT id FROM account WHERE name like 'O'Neal'
```

```
.fromx('account')
.wherex(new dz.FieldCondition('name').likex('O\'Neal'))
.toSoql(new dz.SoqlOptions().escapeSingleQuotes())
//->SELECT id FROM account WHERE name like 'O\'Neal'
```