A short story about configuration file formats

Andreas Rein

January 29, 2016

Motivation

```
INSERT INTO STATUS( STATUS_ID, COLOR )
  SELECT ID, TO_NUMBER( COL, 'xx' ) --Oracle does not support 0x
                                    --to indicate hex numbers
    FROM ( SELECT 1 ID, 'ff0000' COL FROM dual UNION ALL
           SELECT 2   , '00ff00'     FROM dual UNION ALL
           SELECT 3   , 'ffff00'     FROM dual );

INSERT INTO DATASET_TYPE( DATASET_TYPE_ID, FEATURES )
          --FEATURE1 FEATURE2 FEATURE3
  SELECT 1, 1       + 2      + 0        FROM dual UNION ALL
  SELECT 2, 1       + 0      + 4        FROM dual UNION ALL
  SELECT 3, 0       + 2      + 0        FROM dual;


INSERT INTO SERVICE( SERVICE_ID, URL )
  SELECT 1, '192.168.0.1/service1.cgi' FROM dual UNION ALL
  SELECT 2, '192.168.0.1/service2.cgi' FROM dual;
```

During development

```
VAR TEST_SERVER VARCHAR2(50) = '192.168.0.1/';

INSERT INTO SERVICE( SERVICE_ID, URL )
  SELECT 1, TEST_SERVER || 'service1.cgi' FROM dual UNION ALL
  SELECT 2, TEST_SERVER || 'service2.cgi' FROM dual;
```

When deployed

```
INSERT INTO SERVICE( SERVICE_ID, URL )
  SELECT 1, '123.45.67.1/service1.cgi' FROM dual UNION ALL
  SELECT 2, '89.12.34.56/service2.cgi' FROM dual;
```

Candidates

- XML
  Good support in Qt, great when it comes to transformations, complex queries etc. but...
  - A lot of overhead for tabular data
  - Type of a value is not always obvious
    E.g. the value of `<element>42</element>` could be the integer 42 or the string "42"
- INI, CSV
  - No...
- JSON
  - Easy
  - Great for tabular data
  - Basic types are distinguishable
  - Supported by Qt since 5.0 (`QJsonDocument`)

Plain JSON

```
[ { "TableName": "STATUS",
    "Columns": [ "STATUS_ID", "COLOR" ],
     "Data": [ [ 1, "ff0000" ], //JSON also does not support 0x
               [ 2, "00ff00" ],
               [ 3, "ffff00" ] ] },

  { "TableName": "DATASET_TYPE",
    "Columns": [ "DATASET_TYPE_ID", "FEATURES" ],
     "Data": [ [ 1, 3 ], //No mathematical
               [ 2, 5 ], //operations allowed
               [ 3, 2 ] ] },

  { "TableName": "SERVICE",
    "Columns": [ "SERVICE_ID", "URL" ],
     "Data": [ [ 1, "192.168.0.1/service1.cgi" ],
               [ 2, "192.168.0.1/service2.cgi" ] ] } ]
```

Back to the roots: JavaScript

```
return [
  { TableName: 'STATUS',
    Columns: [ 'STATUS_ID', 'COLOR' ],
     Data: [ [ 1, 0xff0000 ], //JavaScript does support 0x...
             [ 2, 0x00ff00 ],
             [ 3, 0xffff00 ] ] },

  { TableName: 'DATASET_TYPE',
    Columns: [ 'DATASET_TYPE_ID', 'FEATURES' ],
              //FEATURE1 FEATURE2 FEATURE3 //..and comments!
     Data: [ [ 1, 1      + 2      + 0 ], //and mathematical
             [ 2, 1      + 0      + 4 ], //operations!
             [ 3, 0      + 2      + 0 ] ] },

  { "TableName": "SERVICE",
    "Columns": [ "SERVICE_ID", "URL" ],
     "Data": [ [ 1, "192.168.0.1/service1.cgi" ],
               [ 2, "192.168.0.1/service2.cgi" ] ] } ];
```

Usage

```
//Must exist before using QJSEngine
QCoreApplication app( argc, argv );

QFile jsFile( "data.js" );
if( !jsFile.open( QIODevice::ReadOnly ) )
{
  //Error handling...
  return;
}

QJSEngine engine;

QJSValue result =
   engine.evaluate( QStringLiteral( "(function(){␣" ) +
                    QString::fromUtf8( jsFile.readAll() ) +
                    QStringLiteral( "})()" ) );
```

Usage

```
//Error reporting
qDebug()
  << QStringLiteral( "Error␣at␣line␣" )
  << result.property( QStringLiteral( "lineNumber" ) ).toString()
  << QStringLiteral( ":␣" )
  << result.property( QStringLiteral( "message" ) ).toString();

//How to get the length of an array
const quint32 tableCount =
    result.property( QStringLiteral( "length" ) ).toUInt();

for( quint32 tableI = 0; tableI < tableCount; ++tableI )
{
  const QJSValue table = result.property( tableI );

  //How to access properties
  const QString tableName =
    table.property( QStringLiteral( "TableName" ) ).toString();

  // ...
}
```

Some more nice features

```
//During development, all files are at the same base path
var currentServer = '192.168.0.1/';

//Easy way to change a specific color system wide
//while still maintaining the possibility to specify
//another one at certain places
var useForGreen = 0x00ff00;

return [
  { TableName: 'STATUS',
    Columns: [ 'STATUS_ID', 'COLOR' ],
     Data: [ [ 1, 0xff0000 ],
             [ 2, useForGreen ],
             [ 3, 0xffff00 ] ] },

  { TableName: 'SERVICE',
    Columns: [ 'SERVICE_ID', 'URL' ],
     Data: [ [ 1, currentServer + 'service1.cgi' ],
             [ 2, currentServer + 'service2.cgi' ] ] } ];
```

Security

- Interpreted code might pose a security risk
- Thus...
  - ...use JavaScript during development
  - ...convert to JSON when deploying (as part of your deployment process)
  - ...use a JSON parser (e.g. QJsonDocument) when deployed
- To omit the need to write two different parsers...

...either go through `QVariant`:

```
#ifdef DEVELOPMENT_MODE
 QVariant result = engine.evaluate(
   QStringLiteral( "(function(){ " ) +
   QString::fromUtf8( jsFile.readAll() ) +
   QStringLiteral( "})()" ) ).toVariant();

  QJsonDocument doc = QJsonDocument::fromVariant( result );
#else
  QJsonDocument doc = QJsonDocument::fromJson( jsFile.readAll() );
#endif

 QJsonArray tableArray = doc.array();

 for( const QJsonValue& tableVal : tableArray )
 {
   const QJsonObject tableObj = tableVal.toObject();

   const QString tableName =
     tableObj.value( QStringLiteral( "TableName" ) ).toString();
   // ...
 }
```

...or use `QVariant` directly:

```
#ifdef DEVELOPMENT_MODE
  QVariant result = engine.evaluate(
    QStringLiteral( "(function(){ " ) +
    QString::fromUtf8( jsFile.readAll() ) +
    QStringLiteral( "})()" ) ).toVariant();
#else
  QVariant result =
    QJsonDocument::fromJson( jsFile.readAll() ).toVariant();
#endif

QVariantList tableArray = result.toList();

for( const QVariant& tableVal : tableArray )
{
  const QVariantMap tableObj = tableVal.toMap();

  const QString tableName =
    tableObj.value( QStringLiteral( "TableName" ) ).toString();
  // ...
}
```

...or use `JSON.stringify`:

```
QByteArray fileData = jsFile.readAll();

#ifdef DEVELOPMENT_MODE
  QJSValue result = engine.evaluate(
    QStringLiteral( "JSON.stringify(␣(function(){␣" ) +
    QString::fromUtf8( jsFile.readAll() ) +
    QStringLiteral( "})()␣)" ) );

  //Error handling

  fileData = result.toString().toUtf8();
#endif

QJsonDocument doc = QJsonDocument::fromJson( fileData );
```

```
QJsonArray tableArray = doc.array();

for( const QJsonValue& tableVal : tableArray )
{
  const QJsonObject tableObj = tableVal.toObject();

  const QString tableName =
    tableObj.value( QStringLiteral( "TableName" ) ).toString();

  //...
}
```

Conclusion

- Broaden your horizon
- Give you the basis for decision making
  Pros:
  - Easy and compact
  - Modern and flexible
  - Safe

  Cons:
  - You need a JavaScript engine (No problem since Qt 5.0)
- Motivate you to consider JavaScript/JSON for your configuration files

Thank you for your attention.