Michigan State University

Node.js Oracle ORM for Kuali Financials

Rob Tucker

05/29/2018

Contents

[Introduction 3](#_Toc515885805)

[Project Layout 4](#_Toc515885806)

[Design Concepts 5](#_Toc515885807)

[Orm 6](#_Toc515885808)

[Model 6](#_Toc515885809)

[Repository 8](#_Toc515885810)

[MetaData 9](#_Toc515885811)

[Test 11](#_Toc515885812)

[MetaData Definition Details 14](#_Toc515885813)

[Field Definition 14](#_Toc515885814)

[Join Definition 14](#_Toc515885815)

[One-to-One Definition 14](#_Toc515885816)

[One-to-Many Definition 15](#_Toc515885817)

[Repository Details 15](#_Toc515885818)

[Join Strategy 15](#_Toc515885819)

[Object Based Database Operations 18](#_Toc515885820)

[Provided Repository Object Operations 19](#_Toc515885821)

[findOne 19](#_Toc515885822)

[find 19](#_Toc515885823)

[exists 19](#_Toc515885824)

[count 19](#_Toc515885825)

[save 20](#_Toc515885826)

[delete 20](#_Toc515885827)

[Provided Repository SQL Operations 20](#_Toc515885828)

[executeSqlQuery 20](#_Toc515885829)

[executeSql 20](#_Toc515885830)

[Repository Operation options Parameter 20](#_Toc515885831)

[WhereComparison.js 21](#_Toc515885832)

[OrderByEntry.js 22](#_Toc515885833)

[Database Configuration 23](#_Toc515885834)

[Application Configuration 24](#_Toc515885835)

[Getting Started 25](#_Toc515885836)

[Some Example Code 25](#_Toc515885837)

[findOne 26](#_Toc515885838)

[find 26](#_Toc515885839)

[count 26](#_Toc515885840)

[exists 27](#_Toc515885841)

[getAll 27](#_Toc515885842)

[save 27](#_Toc515885843)

[delete 28](#_Toc515885844)

[executeSqlQuery 28](#_Toc515885845)

[executeSql 28](#_Toc515885846)

[REST Access 29](#_Toc515885847)

[Appendix A: Code Generator 32](#_Toc515885848)

# Introduction

At MSU we have created a simple JavaScript-based ORM for Kuali Financials on Oracle that runs on Node.js. The application is available as a private package in the npm repository.

The ORM models JPA concepts to some extent and allows Object based database access. For example you can create OQL-like queries such as the example below:

select Account o

from Account

where o.finCoaCd = :finCoaCd and o.accountNbr = :accountNbr

The framework uses the node-oracledb driver for database access and consists a 4 main areas:

* Model
* Repository
* MetaData
* Test

Each standard table in the current KFS and Rice database is represented by Node.js modules from these areas. For example, the CA\_ACCOUNT\_T table has the following objects:

* Account.js – model
* AccountRepository.js – database access
* AccountMetaData.js – metadata definitions
* AccountTest.js – model tests
* AccountRepositoryTest.js – repository tests

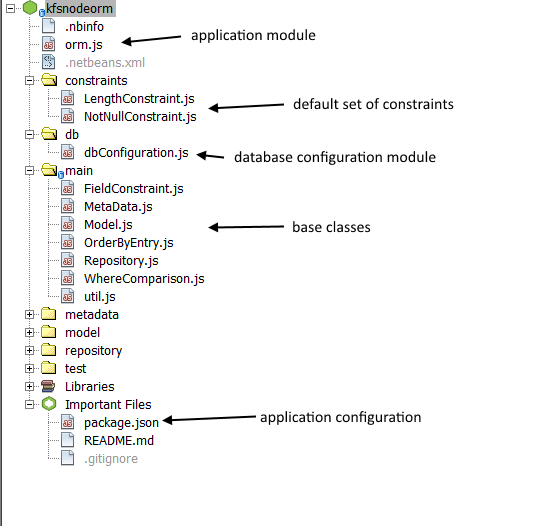
Each repository module supports the following object-based methods out of the box:

* findOne – find object by primary key
* find – find objects by input where parameters
* getAll – return all objects from associated table
* count – count based on where or count all
* exists – object exists in database
* save – performs one or more inserts/updates based on input
* delete – performs one or more delete operations based in input

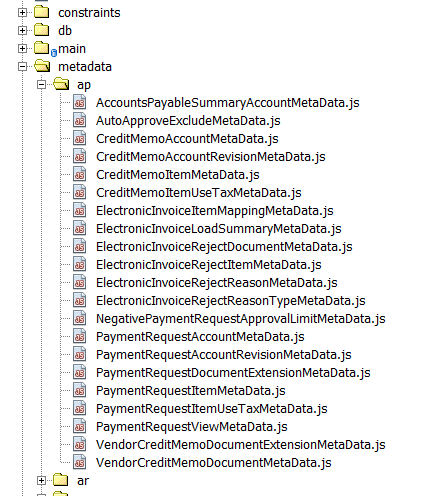
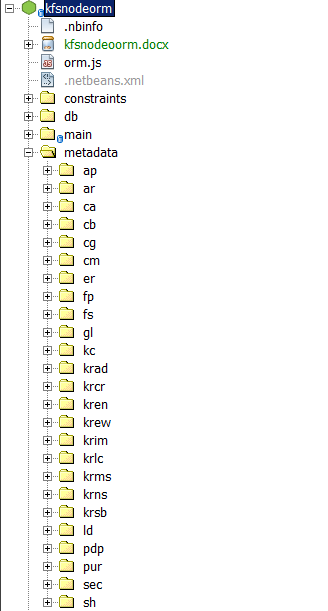
Standard sql and results are also supported. An associated code-generator project exists that will build the startup set of ORM objects based on the OJB definitions found in the KFS and Rice projects. See appendix A for more information on configuring and running the code generator. The project currently is in a pre-beta state. The application functions and basic tests are written and succeeding for the major components but there is still a lot to do.

# Project Layout

The layout of the kfsnodeorm Node.js project is shown below:



The metadata, model, repository and test folders contain the individual table-related modules that were originally created via the code generator mentioned in the introduction, these are broken out in subfolders by the table name prefix:



The kfsnodeorm project is available on github at:

<https://github.com/rbtucker/kfsnodeorm>

# Design Concepts

What we have tried to accomplish in this application is to provide simple database access using the existing KFS and RICE data model while running on Node.js with JavaScript, JSON and enhanced testing capabilities, extensibility, scalability and performance.

As stated in the introduction, the application consists of 4 major areas – model, repository, metadata and test. Each of these areas are built with JavaScript using the [node module system](https://nodejs.org/api/modules.html). A general description of each area can be found below:

## Orm

The orm.js is the application module for kfsnodeorm. The orm initiates the database interface by creating configured database pools then loads all the repository and metadata modules. These modules are made available by model name (all lower case) via the methods below:

orm.getMetaData(‘modelName’);

orm.getRepository(‘modelName’);

The orm also provides database connections from the pool via the orm.getConnection(‘poolAlias’) method.

## Model

A model object is the familiar data container that maps to a database table. Each Model object will extend Model.js to get some basic functionality:

"use strict";

const util = require("./util.js");

class Model {

constructor(metaData) {

this.\_\_model\_\_ = metaData.objectName;

this.metaData = metaData;

this.modified = false;

this.newModel = true;

this.constraintsEnabled = false;

this.data = new Object();

}

isModified() {

return this.modified;

}

A few remarks about the code above:

* “use strict” – enforces strict JavaScript coding, for example variable must be declared with “var” or “let”. If a variable is used and has not been declared an error will be thrown.
* “require” keyword – this is how you import other node modules. In this case I am importing the util.js module
* “class” - the is JavaScript ES6 construct that formalizes the common JavaScript pattern of simulating class-like inheritance hierarchies using functions and prototypes.
* metaData – database metadata definitions associated with this object. When the object is transferred as JSON this information can be removed if desired.
* \_\_model\_\_ - object name of extending class (Account, Chart etc.) with these name we can reconstitute an object from JSON data.
* modified – modified flag that tracks whether object has been updated
* newModel – flag to determine if this is a new model object (not in database).
* constraintsEnabled – model constraints can be enabled (field length checks etc.) . Constraints will be checked when data is set in the object based on the constraintsEnabled flag which is false by default.
* data – this is where the object data values are stored. This is your standard JavaScript object used as a map = data[’fieldName’] = someValue. In the extending classes there will be get/set data access method which allow access to the data – for example getAccountNbr(). These calls just pass through to this storage. This design allows easy access by field name for generic data processing and testing.

Below is a code snippet from the Account.js module:

"use strict";

const Model = require('../../main/Model.js');

class Account extends Model {

constructor(metaData) {

super(metaData);

}

getFinCoaCd() { return this.getFieldValue("finCoaCd"); };

setFinCoaCd(value) { this.setFieldValue("finCoaCd", value); };

getAccountNbr() { return this.getFieldValue("accountNbr"); };

setAccountNbr(value) { this.setFieldValue("accountNbr", value); };

## Repository

The bulk of the work in the ORM goes on in the Repository.js object, this is where the Object-to-SQL logic is carried out and the database access is supported. Repository.js also contains the complex logic for the SQL result set to object graph population. Each repository object will extend the base class Repository.js a portion of which is shown below:

“use strict”

const orm = require('../orm.js');

const util = require('../main/util.js');

const uuid = require(’uuid');

const insertSqlMap = new Map();

const updateSqlMap = new Map();

/\*\*

\* this class is the heart of the orm - all the relations to object graph logic occurs here as well as the sql operations

\*/

module.exports = class Repository {

constructor(poolAlias, metaData) {

this.metaData = metaData;

this.poolAlias = poolAlias;

this.selectedColumnFieldInfo = new Array();

this.columnPositions = new Array();

this.pkPositions = new Array();

this.namedDbOperations = new Map();

this.generatedSql = new Array();

// default named db operations

this.namedDbOperations.set(util.FIND\_ONE, this.buildFindOneNamedOperation(metaData));

this.namedDbOperations.set(util.GET\_ALL, this.buildGetAllNamedOperation(metaData));

this.selectClauses = new Array();

this.joinClauses = new Array();

A few comments about the code above:

* const orm – this is the kfsnodeorm application module. When the application starts it builds maps of all repositories and metaData modules which can be accessed via the orm.getMetaData(“objectName”) and orm.getRepository(“objectName”) calls.
* insertSqlMap, updateSqlMap – these are here for performance reasons, since each object has a distinct insert and update statement it makes no sense to generate those sql statements on every call. If the statement is not in the map it will be generated and added to map as required.
* poolAlias – each extending repository object will have an associated pool alias. A pool is created for each database connection that is configured. In this way we can run multiple database connections and access different databases based on the alias – KFS and RICE for example.
* Other maps and arrays defined – most of these are created for performance . There is a lot of work when converting 2 dimensional sql result set to an object graph so we try to cache as much of this information as we can – for example, the primary key positions for various joined tables in the result rows.
* namedDbOperations – as stated in the introduction, pre-defined object queries can be defined and used – they will end up in this map. As you can see findOne and getAll object queries are created by default. Extending repository classes can override the loadNamedDbOperations() to add custom object queries.

Below is the AccountRepository.js module:

"use strict";

const orm = require('../../orm.js');

const poolAlias = 'kfsdb';

const Repository = require('../../main/Repository.js');

class AccountRepository extends Repository {

constructor(metaData) {

super(poolAlias, metaData);

};

loadNamedDbOperations() {

// define named database operations here

// - the convention is as follows

// namedDbOperations.set('functionName', 'objectQuery')

// example: select Account o from Account

// where o.finCoaCd = :finCoaCd and o.accountNbr := accountNbr

};

};

module.exports = function(metaData) {

return new AccountRepository(metaData);

};

## MetaData

The metadata objects contains object to/from database definitions as well as the object relationship definitions. Each MetaData object will extend the base class MetaData.js:

/\*\*

\* this is the base class for defing the sql to object mapping definitions

\*/

class MetaData {

constructor(

objectName,

module,

tableName,

fields,

oneToOneDefinitions,

oneToManyDefinitions) {

this.objectName = objectName;

this.module = module;

this.tableName = tableName;

this.fields = fields;

this.oneToOneDefinitions = oneToOneDefinitions;

this.oneToManyDefinitions = oneToManyDefinitions;

this.fieldConstraints = new Map();

// map of column name to field definitions

this.columnToFieldMap = new Map();

// map of field name to field definitions

this.fieldMap = new Map();

// add some default constraints –

// will be disabled by default in the model object

for (let i = 0; i < fields.length; ++i) {

this.columnToFieldMap.set(fields[i].columnName, fields[i]);

this.fieldMap.set(fields[i].fieldName, fields[i]);

if (fields[i].required) {

let l = null;

if (!this.fieldConstraints.has(fields[i].fieldName)) {

l = new Array();

this.fieldConstraints.set(fields[i].fieldName, l);

} else {

l = this.fieldConstraints.get(fields[i].fieldName);

}

l.push(new (require("../constraints/NotNullConstraint.js")));

}

if (this.isLengthConstraintRequired(fields[i])) {

let l = null;

if (!this.fieldConstraints.has(fields[i].fieldName)) {

l = new Array();

this.fieldConstraints.set(fields[i].fieldName, l);

} else {

l = this.fieldConstraints.get(fields[i].fieldName);

}

l.push(new (require("../constraints/LengthConstraint.js"))

(this.getMaxLength(fields[i])));

}

this.loadConstraints();

}

}

As you can see this is a generic class that stores the definitions. The extending class will pass in definitions in their constructor with the super() call. Also note that in this base class we are creating LengthConstraint and a NotNullConstraint for fields if required. These will be checked when data is set on the model if the model.enableConstraints flag is true. Custom constraints can be added in extending classes by overriding the loadConstraints() method.

A portion of the AccountMetaData.js module is shown below:

"use strict";

var MetaData = require('../../main/MetaData.js').MetaData;

class AccountMetaData extends MetaData {

constructor() {

super(

'Account', // object name,

'model/ca/Account.js', // relative module path,

'CA\_ACCOUNT\_T', // table name

[ // field definitions - order is important,

//selected data will be in this order,

// primary key fields shoud be first

{ // 0

fieldName: "finCoaCd",

type: "VARCHAR(2)",

columnName: "FIN\_COA\_CD",

required: true,

primaryKey: true

},

{ // 1

fieldName: "accountNbr",

type: "VARCHAR(32)",

columnName: "ACCOUNT\_NBR",

required: true,

primaryKey: true

},

{ // 2

fieldName: "acCstmIcrexclCd",

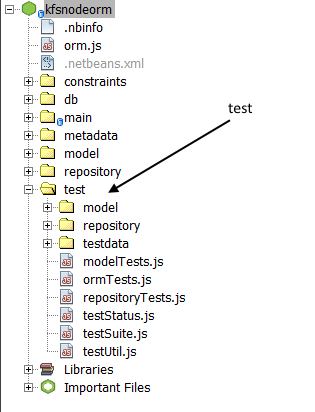
type: "VARCHAR(1)",

columnName: "AC\_CSTM\_ICREXCL\_CD"

},

## Test

Below is the test hierarchy in the project:



If the appconfig.json testMode is set to true then tests will be run after ORM startup. The entry point for all tests is testSuite.js.

"use strict";

const util = require("../main/util.js");

const ormTests = module.require("./ormTests.js");

const modelTests = module.require("./modelTests.js");

const repositoryTests = module.require("./repositoryTests.js");

const assert = require("chai").assert;

const orm = require("../orm.js");

module.exports.run = async function() {

util.logInfo("running testSuite...");

try {

util.logInfo("running orm tests...");

await ormTests.run(orm);

util.logInfo(" - orm tests complete");

util.logInfo("running model tests...");

await modelTests.run(orm);

util.logInfo(" - model tests complete");

util.logInfo("running repository tests...");

await repositoryTests.run(orm);

util.logInfo(" - repository tests complete");

util.logInfo("testSuite complete");

}

catch (e) {

util.logError(e.message);

}

};

Currently there are basic tests for orm.js and each Model and Repository object – these tests can be found under the test/model and test/repository folders. A portion of the AccountTest.js (for model) is shown below:

"use strict";

const orm = require('../../../orm.js');

const testUtil = require('../../testUtil.js');

const util = require('../../../main/util.js');

const assert = require(‘chai').assert;

module.exports.run = function(metaData) {

let fields = metaData.getFields();

// test basic data handling

let model = testDataHandling(metaData, fields);

// model should be modified now

assert(model.isModified(), 'expected model to be modified but is not');

// test constraint handling

model.enableConstraints(true);

testConstraints(model, fields);

model.enableConstraints(false);

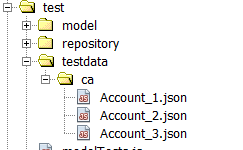
};

Obviously the repository tests are much more involved. The majority of the tests are handled dynamically by accessing a configured test database. Database connection information is setup in a json file outside of the application and loaded at startup. The file locations are specified in the appconfig.json as shown below:

"dbConfiguration" : "c:\\work\\kfsormsec.json",

"testDbConfiguration" : "c:\\work\\kfsormsec.json",

Updates and queries are easily tested in a dynamic fashion as long as there is data in the test database. Inserts are not so easy. The test design expects test json files for insert tests. These files are just the json version of the model objects with modified data that can be inserted. The files can be found under the testdata folder:



and have a naming convention of - <ObjectName>\_index.json

For example Account\_1.json, Account\_2.json …

# MetaData Definition Details

The metadata definitions are crucial to the operation of the ORM. Database queries are generated as a join based on the metadata definitions. The depth of the resulting object hierarchy – and the complexity of the required join to get this depth is driven by the maxDefaultJoinDepth setting in appconfig.json.

## Field Definition

Below is an example field definition from AccountMetaData.js:

{ // 0

fieldName: "finCoaCd",

type: "VARCHAR(2)",

columnName: "FIN\_COA\_CD",

required: true,

primaryKey: true

},

The fieldname, type and columnName are required. Order of the field definitions is important. It is expected that the primary key fields will be listed first then the remaining fields. Select order will be the same as the field definition order. Available field definition names are list below:

* fieldname – field name to be used on the object
* type – database type
* columnName – database column name
* required – true/false – true mean not nullable
* primaryKey – true if primary key field
* versionColumn – true if this column should be used for optimistic locking
* defaultValue – default value on insert if no entry
* sequence – sequence name to use for this column on insert

## Join Definition

Join definitions define parent-child relationships between table. At this stage 2 types are supported one-to-one and one-to-many. Below is an example of each from the AccountMetaData.js module:

### One-to-One Definition

{ // 0

fieldName: "chartOfAccounts",

targetModelName: "Chart",

targetModule: "../model/ca/Chart.js",

targetTableName: "CA\_CHART\_T",

status: "enabled",

joinColumns : {

sourceColumns : "FIN\_COA\_CD",

targetColumns : "FIN\_COA\_CD"

}

},

### One-to-Many Definition

{ // 0

fieldName: "subAccounts",

targetModelName: "SubAccount",

targetModule: "../model/ca/SubAccount.js",

targetTableName: "CA\_SUB\_ACCT\_T",

cascadeUpdate: true,

cascadeDelete: true,

status: "enabled",

joinColumns : {

sourceColumns : "FIN\_COA\_CD,ACCOUNT\_NBR",

targetColumns : "FIN\_COA\_CD,ACCOUNT\_NBR"

}

},

The logic in these definitions differs from the JPA annotation logic in that the terms target and source are based on the current metadata object regardless of the relationship so if I am in the AccountMetaData.js module source always refers to the Account and target refers to the related object. fieldname, targetModelName, targetModule, targetTableName, status and joinColumns are required. Below are the available field descriptions:

* fieldName – field name for this relationship instance on the parent object
* targetModelName – object name for target module
* targeModule – relative path to target module definition file
* targetTableName – table name of related table
* cascadeUpdate – true if this relationship depends upon parent
* cascadeDelete – true if related object from this relationship should be deleted when parent is deleted
* status – can be one of 3 values: enabled,disabled,lazyload – if enabled then sql join will be created if this relationship falls within the join depth specified. If disabled, this relationship will be ignored for join purposes. Lazyload is currently not implemented but eventually will support deferred load when required.
* required – true/false – if true an inner join will be generated else left outer join will be generated
* joinColumns – json that contains comma-delimited lists for source-to-target join specification

# Repository Details

## Join Strategy

The bulk of the work in the ORM goes on in the Repository.js base class. When running queries SQL is generated as one select with joins to related tables as defined by the one-to-one and one-to-many definitions in the associated metadata. The "defaultMaxJoinDepth" entry in appconfig.json defines how deep the repository will traverse down the parent/child hierarchy and create joins – the deeper you go, the bigger the select statement and the poorer the performance. The top level class, always designated with alias “t0” will be the only table to join one-to-one child to parent relationships, the rest of the joins will be parent to child on the one-to-one and one-to-many definitions. The repository object supports running a query with a join depth specified at any level from 0 to the defaultMaxJoinDepth. A join depth of 0 is a special case where only the column data for the top-level object is pulled– no joins are created. If you need a high performance query on a large table and you do not need related information you can run the query with joinDepth = 0. If no specific joinDepth is specified for a query the the defaultMaxJoinDepth will be used.

Below is the sql generated for a findOne call on the ChartRepository with no joinDepth specified and defaultMaxJoinDepth = 3.

SELECT t0.FIN\_COA\_CD,

t0.EXPBDGT\_ELIMOBJ\_CD,

t0.FIN\_AP\_OBJ\_CD,

t0.FIN\_AR\_OBJ\_CD,

t0.FIN\_CASH\_OBJ\_CD,

t0.FIN\_COA\_ACTIVE\_CD,

t0.FIN\_COA\_DESC,

t0.FIN\_EXT\_ENC\_OBJ\_CD,

t0.FIN\_INT\_ENC\_OBJ\_CD,

t0.FIN\_PRE\_ENC\_OBJ\_CD,

t0.FND\_BAL\_OBJ\_CD,

t0.ICR\_EXP\_FIN\_OBJ\_CD,

t0.ICR\_INC\_FIN\_OBJ\_CD,

t0.INCBDGT\_ELIMOBJ\_CD,

t0.OBJ\_ID,

t0.RPTS\_TO\_FIN\_COA\_CD,

t0.VER\_NBR,

t325\_0.FIN\_COA\_CD,

t325\_0.EXPBDGT\_ELIMOBJ\_CD,

t325\_0.FIN\_AP\_OBJ\_CD,

t325\_0.FIN\_AR\_OBJ\_CD,

t325\_0.FIN\_CASH\_OBJ\_CD,

t325\_0.FIN\_COA\_ACTIVE\_CD,

t325\_0.FIN\_COA\_DESC,

t325\_0.FIN\_EXT\_ENC\_OBJ\_CD,

t325\_0.FIN\_INT\_ENC\_OBJ\_CD,

t325\_0.FIN\_PRE\_ENC\_OBJ\_CD,

t325\_0.FND\_BAL\_OBJ\_CD,

t325\_0.ICR\_EXP\_FIN\_OBJ\_CD,

t325\_0.ICR\_INC\_FIN\_OBJ\_CD,

t325\_0.INCBDGT\_ELIMOBJ\_CD,

t325\_0.OBJ\_ID,

t325\_0.RPTS\_TO\_FIN\_COA\_CD,

t325\_0.VER\_NBR

FROM CA\_CHART\_T t0

LEFT OUTER JOIN CA\_CHART\_T t325\_0 ON (t325\_0.FIN\_COA\_CD = t0.RPTS\_TO\_FIN\_COA\_CD)

WHERE t0.FIN\_COA\_CD = :finCoaCd

Here is the same call with joinLevel = 0;

SELECT t0.FIN\_COA\_CD,

t0.EXPBDGT\_ELIMOBJ\_CD,

t0.FIN\_AP\_OBJ\_CD,

t0.FIN\_AR\_OBJ\_CD,

t0.FIN\_CASH\_OBJ\_CD,

t0.FIN\_COA\_ACTIVE\_CD,

t0.FIN\_COA\_DESC,

t0.FIN\_EXT\_ENC\_OBJ\_CD,

t0.FIN\_INT\_ENC\_OBJ\_CD,

t0.FIN\_PRE\_ENC\_OBJ\_CD,

t0.FND\_BAL\_OBJ\_CD,

t0.ICR\_EXP\_FIN\_OBJ\_CD,

t0.ICR\_INC\_FIN\_OBJ\_CD,

t0.INCBDGT\_ELIMOBJ\_CD,

t0.OBJ\_ID,

t0.RPTS\_TO\_FIN\_COA\_CD,

t0.VER\_NBR

FROM CA\_CHART\_T t0

WHERE t0.FIN\_COA\_CD = :finCoaCd

JSON results from a findOne() call with joinDepth = 3 will be something like this:

{

"\_\_model\_\_": "Chart",

"modified": false,

"newModel": false,

"constraintsEnabled": false,

"data": {

"finCoaCd": "MS",

"expbdgtElimobjCd": "4054",

"finApObjCd": "2055",

"finArObjCd": "1129",

"finCashObjCd": "1100",

"finCoaActiveCd": "Y",

"finCoaDesc": "MICHIGAN STATE UNIVERSITY",

"finExtEncObjCd": "3042",

"finIntEncObjCd": "3043",

"finPreEncObjCd": "3040",

"fndBalObjCd": "3037",

"icrExpFinObjCd": "6487",

"icrIncFinObjCd": "4065",

"incbdgtElimobjCd": "4054",

"objId": "777B41D2001DEFB8E040007F01003339",

"rptsToFinCoaCd": "MA",

"verNbr": 4,

"reportsToChartOfAccounts": {

"\_\_model\_\_": "Chart",

"modified": false,

"newModel": false,

"constraintsEnabled": false,

"data": {

"finCoaCd": "MA",

"expbdgtElimobjCd": "4054",

"finApObjCd": "2055",

"finArObjCd": "1003",

"finCashObjCd": "1100",

"finCoaActiveCd": "Y",

"finCoaDesc": "MICHIGAN STATE UNIVERSITY ADMIN",

"finExtEncObjCd": "3042",

"finIntEncObjCd": "3043",

"finPreEncObjCd": "3040",

"fndBalObjCd": "3037",

"icrExpFinObjCd": "6487",

"icrIncFinObjCd": "4065",

"incbdgtElimobjCd": "4054",

"objId": "777B41D2001EEFB8E040007F01003339",

"rptsToFinCoaCd": "MA",

"verNbr": 5

}

}

}

}

## Object Based Database Operations

Repository data access is designed to use object-based queries. An example of a typical query is shown below:

select Account o where from Account

where o.acctClosedInd = ‘N’

order by o.finCoaCd, o.accountNbr

The “o.” prefix shown above is significant and is required in the OQL. Field names can be specified using dot notation foe example:

select Account o where from Account

where o.acctClosedInd = ‘N’ and o.subAccounts.subAcctNm = ‘someName’

order by o.finCoaCd, oaccountNbr

You can add predefined sql operations that can be called up and used by the repository by overriding the

Repository.loadNamedDbOperations() method and adding the custom operations as shown below:

loadNamedDbOperations() {

getNamedDbOperations().add(“findByName”,

“select Account o from Account where o.accountNm = :accountNm’);

}

You can now execute the query using the method Repository. executeNamedDbOperation() as shown below:

Let params = new Array();

params.push(‘myAccountName’)

let repo = orm.getRepository(‘Account’);

let result = await repo.executeNamedDbOperation(‘findByName’, params);

## Provided Repository Object Operations

The base class Repository.js provides the database operations described below out of the box. In all cases, if an error occurs a json object of the form {error: <error info> } will be returned.

### findOne

The findOne method returns a model object by primary key – the signature is shown below:

async findOne(primaryKey, options)

* primaryKey – required array of the primary key values for bind parameters. The order is important, values should be in the order of the field definitions in the metadata.
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

### find

Returns an array of model objects matching the criteria passed to the method

async find(whereComparisons, orderByEntries, options)

* whereComparisons – required array of WhereComparison.js objects defining the where clause
* orderByEntries – optional array of OrderByEntry.js defining the order by clause, if not included the query will be order by primary key values
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

### exists

Returns true if input parameter exists in database, false otherwise

async exists(modelInstance, options)

* modelInstance – required - model to check
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

### count

Returns count of objects matching input criteria. If no input where clause return table row count

async count(whereComparisons, options)

* whereComparisons – optionall array of WhereComparison.js objects defining the where clause
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

### save

Updates or inserts new model records to the database. The entire object graph will be persisted based on the metadata relationship configuration (cascadeUpdate).

async save(modelInstances, options)

* modelInstances – required – one or more model instances to save
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

### delete

Deletes data from the database matching model instances passed in the input. The entire object graph will be processed based on the metadata relationship configuration (cascadeDelete).

async delete(modelInstances, options)

* modelInstances – required – one or more model instances to delete
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

## Provided Repository SQL Operations

The base class Repository.js supports standard SQL operations described below. In all cases, if an error occurs a json object of the form {error: <error info> } will be returned.

### executeSqlQuery

Executes standard sql query and returns result in row/column format

async executeSqlQuery(sql, parameters, options)

* sql – required sql select statement
* parameters – optional array of bind parameter values
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

### executeSql

Executes a non-select sql statement

async executeSql(sql, parameters, options)

* sql – required sql select statement
* parameters – optional array of bind parameter values
* options – optional parameter that specifies additional information – see the next section for more information. If no options parameter is passed a default will be created and used.

## Repository Operation options Parameter

As described above, all of the base sql operations can take an “options” argument. This parameter is a basic json object that adds additional parameters to the standard oracledb options. The oracledb options are listed below and you can find a detailed description here [https://github.com/oracle/node-oracledb/blob/master/doc/api.md#executeoptions](https://github.com/oracle/node-oracledb/blob/master/doc/api.md%23executeoptions)

* autoCommit – defaults to false
* extendedMetaData – defaults to false
* fetchArraySize
* fetchInfo
* maxRows
* outFormat
* prefetchRows
* resultSet

The custom options are all optional and are described below:

* distinct – true/false – if set to true on a query will add “distinct” to the select clause
* conn – by default all sql operations pull a connection from the configured connection pool which will rollback on close if no commit is executed. In order to handle multi-operation transactional processing you can pull a connection using the orm.getConnection(poolAlias) method and pass this connection to all operations via the options parameter. If a connection exists in the incoming options parameter the methods described above will use this connection and will not issue a close().
* joinDepth – by default all the sql operations will construct sql based on the maxDefaultJoinDepth value from the appconfig.json. If you want to override this setting for a particular call you can set the desired joinDepth in the options parameter.
* returnValues – true/false – by default the save method only returns a “rowsAffected” count, if you would like to have the actual updated records returned set this to true. The will be return in result.updatedValues.

## WhereComparison.js

The WhereComparison module defines where clause comparison information. An array of WhereComparison objects will be passed to repository methods to generate the desired where clause. A portion of the source is shown below:

class WhereComparison {

constructor(fieldName, comparisonValue, comparisonOperator, logicalOperator, useBindParams) {

this.fieldName = fieldName;

this.comparisonValue = comparisonValue;

this.comparisonOperator = comparisonOperator;

this.openParen = '';

this.closeParen = '';

if (util.isDefined(logicalOperator)) {

this.logicalOperator = logicalOperator;

} else {

this.logicalOperator = util.AND;

}

if (util.isDefined(useBindParams)) {

this.useBindParams = useBindParams;

} else {

this.useBindParams = true;

}

if (this.isUnaryOperator()) {

this.useBindParams = false;

comparisonValue = '';

}

if (util.IN === comparisonOperator) {

this.useBindParams = false;

}

}

Below are the field descriptions:

* fieldName – object field name of the field that this comparison applies to. Dot notation can be used for child objects, for example on the Account objects subAccounts.subAcctNm.
* comparisonValue – this is the value to compare. By default the generated where clause will use bind parameters and pass the comparison values in a parameter list
* comparisonOperator – supports standard sql comparisons: =, >=, <= <>, is null, is not null, like and in. The util module includes constants for these values:

const EQUAL\_TO = '=';

const GREATER\_THAN = '>';

const LESS\_THAN = '<';

const LEES\_THAN\_OR\_EQ = '<=';

const GREATER\_THAN\_OR\_EQ = '>';

const NOT\_EQUAL = '<>';

const LIKE = 'like';

const IN = 'in';

const NOT\_NULL = 'is not null';

const NULL = 'is null';

* logicalOperator – and/or – defaults to and – the util module includes constants for these values

const AND = 'and';

const OR = 'or';

* openParen and closeParen – these allow one or more parenthesis to be added at the beginning or end of the generated comparison.
* useBindParams – true/false – defaults to true except in the case where the comparisonOperator is “in”. The in clause is handled in special manner. It is expected that the comparison value will be an array of values and the bind parameter flag will be ignored for the in clause.

## OrderByEntry.js

The OrderByEntry module is used to generate the order by clause. An array of these values will be passed to the repository method to generate the order by clause. A portion of the source is shown below:

class OrderByEntry {

constructor(fieldName, descending) {

this.fieldName = fieldName;

this.descending = descending;

}

getFieldName() {

return this.fieldName;

}

isDescending() {

return this.descending;

}

}

Below are the field descriptions

* fieldName – object field name of the field that this comparison applies to. Dot notation can be used for child objects, for example on the Account objects subAccounts.subAcctNm.
* descending – true/false – defaults to false

# Database Configuration

Datasource and connection pool initialization logic is provided in the class dbConfiguration.js.

"use strict";

const oracledb = require(‘oracledb’);

const util = require("../main/util.js");

const fs = require('fs');

module.exports = function(poolCreatedEmitter, appConfiguration) {

if (testMode) {

initPool(appConfiguration.testDbConfiguration,

poolCreatedEmitter);

} else {

initPool(appConfiguration.dbConfiguration,\

poolCreatedEmitter);

}

};

async function initPool(securityPath, poolCreatedEmitter) {

util.logInfo("creating connection pools...");

let pdefs = JSON.parse(fs.readFileSync(securityPath));

for (let i = 0; i < pdefs.pools.length; ++i) {

await oracledb.createPool(pdefs.pools[i]).then(function(pool) {

util.logInfo(" " + pool.poolAlias

+ " connection pool created");

});

}

// tell orm init that pools are created

poolCreatedEmitter.emit("poolscreated");

}

Multiple connection pools can be created and used by the application. Each connection pool is defined by a pool alias and each repository object has an assigned pool alias. The database connection parameters are found outside the application in a json file specified by the appconfig.json entries:

"dbConfiguration" : "c:\\work\\kfsormsec.json",

"testDbConfiguration" : "c:\\work\\kfsormsec.json",

Below is an example:

{

"pools": [{

"user": "FINDEV",

"password": "mypass",

"connectString": "localhost:1521/mykfsdb",

"poolAlias": "kfsdb",

"poolMax": 20,

"poolMin": 2,

"poolIncrement": 5,

"poolTimeout": 120,

"retryCount": 3,

"retryInterval": 500,

"runValidationSQL": true,

"validationSQL": "alter session set current\_schema=FINDEV"

},

{

"user": "RICE",

"password": "mypass",

"connectString": "localhost:1521/myricedb",

"poolAlias": "ricedb",

"poolMax": 20,

"poolMin": 2,

"poolIncrement": 5,

"poolTimeout": 120,

"retryCount": 3,

"retryInterval": 500,

"runValidationSQL": true,

"validationSQL": "alter session set current\_schema=RICE"

}]

}

# Application Configuration

Below is the appconfig.json file for the kfsnodeorm Node.js application.

{

"testMode" : true,

"stopTestsOnFailure" : false,

"dbConfiguration" : "c:\\work\\kfsormsec.json",

"testDbConfiguration" : "c:\\work\\kfsormsec.json",

"defaultMaxJoinDepth" : 3,

"startRestServer" : false,

"aliases" : {

"vcmdocext" : "vendorcreditmemodocumentextension",

"nprapproveallimit" : "negativepaymentrequestapprovallimit"

}

}

The entries are described below:

* testMode – if set to true, tests will be run after the application is loaded
* stopTestsOnFailure – if set to true tests will stop executing after a failure, otherwise the remaining tests will continue to run
* dbConfiguration – the is the location of database connection information for the non-test database
* testDbConfiguration – location of the connection information for the test database
* defaultMaxJoinDepth – the is how far down the parent/child table hierarchy that joins will be created by default
* startRestServer – if set to true then REST access will be made available to ORM via a REST server
* aliases – this is a list of aliases -> model names to allow using shorter names as url parameters in the rest calls. When a rest call is made the model and the method will be passed as part of the url. By default repository and metadata information is stored as the model name in all lower case – for example “account” for long model names such as “negativepaymentrequestapprovallimit” this can be tedious so you can define an alias that can be used on the URL.

# Getting Started

Below are the steps to setup Node.js for using kfsnodeorm. I used this setup to run from my IDE and hit the Node installation. If you just want to run the app you can do the standard **npm install kfsnodeorm** if you have access to the npm package.

1. Install Node.js version 9.11 or greater
2. Install the following modules:

npm install oracledb – oracle node driver

npm install winston - logging

npm install chai - testing

npm install events – event support

npm install properties-reader

npm install uuid – guid generation

npm install kfsnodeorm

1. Create database connection json file
2. Create/update appconfig.json and set the application properties discussed in previous section.

# Some Example Code

The code below will use the Account object :

## findOne

let repo = orm.getRepositry(‘Account’);

let params = new Array();

params.push(‘MS’);

params.push(‘MYACCOUNTNBR’);

let res = repo.findOne(params);

if (util.isDefined(res.error)) {

// handle error

} else {

return res.result;

}

## find

let repo = orm.getRepositry(‘Account’);

let whereComparisons = new Array();

whereComparisons.push(require('../main/WhereComparison.js')(‘finsCoaCd’, ‘MS’, util.EQUAL\_TO));

whereComparisons.push(require('../main/WhereComparison.js')(‘ acctClosedInd’, ‘Y’, util.EQUAL\_TO));

let orderByEntries = new Array();

orderByEntries.push(require('../main/OrderByEntry.js')(‘ accountNbr’);

let res = repo.find(whereComparions, orderByEntries);

if (util.isDefined(res.error)) {

// handle error

} else {

return res.result;

}

## count

let repo = orm.getRepositry(‘Account’);

let whereComparisons = new Array();

whereComparisons.push(require('../main/WhereComparison.js')(‘finsCoaCd’, ‘MS’, util.EQUAL\_TO));

whereComparisons.push(require('../main/WhereComparison.js')(‘ acctClosedInd’, ‘Y’, util.EQUAL\_TO));

let res = repo.count(whereComparions);

if (util.isDefined(res.error)) {

// handle error

} else {

return res.result;

}

## exists

let repo = orm.getRepositry(‘Account’);

let model = orm.newModelInstance(orm.getMetaData(‘Account’);

model.setFinCoaCd(‘MS’);

model.setAccountNbr(‘1234567’);

let res = repo.exists(model);

if (util.isDefined(res.error)) {

// handle error

} else {

return res.result;

}

## getAll

let res = orm.getRepositry(‘Account’).getAll();

if (util.isDefined(res.error)) {

// handle error

} else {

return res.result;

}

## save

let repo = orm.getRepositry(‘Account’);

let conn = orm.getConnection(repo.getAlias());

let params = new Array();

params.push(‘MS’);

params.push(‘MYACCOUNTNBR’);

let model = repo.findOne(params, {conn: conn});

model.setAcctClosedInd(‘Y’);

let res = repo.save(model, {conn: conn});

if (util.isDefined(res.error)) {

conn.rollback();

// handle error

} else {

conn.commit():

return res.result;

}

conn.close();

## delete

let repo = orm.getRepositry(‘Account’);

let conn = orm.getConnection(repo.getAlias());

let params = new Array();

params.push(‘MS’);

params.push(‘MYACCOUNTNBR’);

let model = repo.findOne(params, {conn: conn});

model.setAcctClosedInd(‘Y’);

let res = repo.delete(model, {conn: conn});

if (util.isDefined(res.error)) {

conn.rollback();

// handle error

} else {

conn.commit():

return res.result;

}

conn.close();

## executeSqlQuery

let repo = orm.getRepositry(‘Account’);

repo.executeSqlQuery(‘select ACCOUNT\_NM from CA\_ACCOUNT\_T where ACCT\_CLOSED\_IND = ‘Y’ order by 1’);

if (util.isDefined(res.error)) {

// handle error

} else {

return res.result;

}

## executeSql

let repo = orm.getRepositry(‘Account’);

let conn = orm.getConnection(repo.getAlias());

repo.executeSql(‘update CA\_ACCOUNT\_T set ACCT\_CLOSED\_IND = ‘N’ where ACCOUNT\_CLOSED\_IND = ‘Y’, [], {conn: conn});

if (util.isDefined(res.error)) {

conn.rollback();

// handle error

} else {

conn.commit();

return res.result;

}

conn.close();

# REST Access

Basic REST access is available with the application using the Node express package. To run rest server on startup set the startRestServer=true in appconfig.json. Within orm.js there are a couple of constants the setup some of the REST configuration:

const REST\_URL\_BASE = '/kfsorm';

const REST\_SERVER\_PORT = process.env.KFS\_ORM\_REST\_PORT || 8888;

You can see the startup in the code below:

var poolCreatedEmitter = new events.EventEmitter();

poolCreatedEmitter.on("poolscreated", async function() {

loadOrm();

if (appConfiguration.testMode) {

let suite = require("./test/testSuite.js");

await suite.run();

}

if (appConfiguration.startRestServer) {

startRest();

}

});

A code snippet of the server code is shown below:

function startRest() {

util.logInfo('starting ' + APP\_NAME + ' REST server...');

server.listen(REST\_SERVER\_PORT, () => {

util.logInfo(APP\_NAME + ' is live on port ' + REST\_SERVER\_PORT);

});

server.get(REST\_URL\_BASE + '/:module/:method', async function(req, res) {

let repo = repositoryMap.get(req.params.module);

let md = metaDataMap.get(req.params.module);

let params = new Array();

let pk = md.getPrimaryKeyFields();

let fields = md.getFields();

if (util.isUndefined(repo) || util.isUndefined(md)) {

res.status(400).send('invalid module \''

+ req.params.module + '\' specified');

} else {

var result;

switch(req.params.method.toLowerCase()) {

case util.FIND\_ONE.toLowerCase():

for (let i = 0; i < pk.length; ++i) {

params.push(req.query[pk[i].fieldName]);

}

result = await repo.findOne(params);

break;

case util.FIND.toLowerCase():

for (let i = 0; i < fields.length; ++i) {

if (util.isDefined(req.query[fields[i].fieldName])) {

params.push(require('./main/WhereComparison.js')

(fields[i].fieldName,

req.query[fields[i].fieldName], util.EQUAL\_TO));

}

}

result = await repo.find(params);

break;

case util.COUNT.toLowerCase():

for (let i = 0; i < fields.length; ++i) {

if (util.isDefined(req.query[fields[i].fieldName])) {

params.push(require('./main/WhereComparison.js')

(fields[i].fieldName,

req.query[fields[i].fieldName], util.EQUAL\_TO));

}

}

result = await repo.count(params);

break;

case util.EXISTS.toLowerCase():

for (let i = 0; i < pk.length; ++i) {

params.push(req.query[pk[i].fieldName]);

}

result = await repo.exists(params);

break;

default:

res.status(400).send('invalid method \''

+ req.params.method + '\' specified');

break;

}

if (util.isUndefined(result)) {

res.status(404).send('not found');

} else if (util.isDefined(result.error)) {

res.status(500).send(util.toString(result.error));

} else if (util.isDefined(result.result)) {

res.status(200).send(util.toDataTransferString(result.result));

} else {

res.status(200).send(result);

}

}

res.end();

});

server.post(REST\_URL\_BASE + '/:module/:method', async function(req, res) {

let repo = repositoryMap.get(req.params.module);

let md = metaDataMap.get(req.params.module);

if (util.isUndefined(repo) || util.isUndefined(md)) {

res.status(400).send('invalid module \''

+ req.params.module + '\' specified');

} else {

let result;

switch(req.params.method.toLowerCase()) {

case util.FIND\_ONE.toLowerCase():

result = await repo.findOne(req.query.primaryKeyValues);

break;

case util.FIND.toLowerCase():

result = await

repo.find(populateWhereFromRequestInput(

req.query.whereComparisons),

populateOrderByFromRequestInput(

req.query.orderByEntries), req.query.options);

break;

case util.SAVE.toLowerCase():

result = repo.save(req.query.modelInstances,

req.query.options);

break;

default:

res.status(400).send('invalid method \''

+ req.params.method + '\' specified');

break;

}

if (util.isUndefined(result)) {

res.status(404).send('not found');

} else if (util.isDefined(result.error)) {

res.status(500).send(util.toString(result.error));

} else if (util.isDefined(result.result)) {

res.status(200).

send(util.toDataTransferString(result.result));

} else if (util.isDefined(result.updatedValues)) {

res.status(200).

send(util.toDataTransferString(result.updatedValues));

} else if (util.isDefined(result.rowsAffected)) {

res.status(200).send(util.toDataTransferString(result));

} else {

res.status(200).send(result);

}

}

res.end();

});

As you can see this API supports the base calls supported by the repository – findOne, find, exists, count, save and delete. The URL used is something like:

http://localhost:8888/kfsorm/account/findOne?finCoaCd=MS&accountNbr=RA123456

where “account” is the ORM object name (all lowercase) and “findOne” is the method name (not case sensitive). Simple GET calls for findOne, find, exists and count are supported as shown. It is assumed that the where clause will be all ands. More complex queries as well as save and delete are supported by POST, DELETE and PUT. In the POST query you can pass and array of WhereComparison objects and OrderByEntry objects for a more complex query definition.

# Appendix A: Code Generator

There is a java-based code generator to create the initial ORM objects from OJB definition files. This project is in a private repository found at:

<https://github.com/rbtucker/kfscore2nodejs.git>

The code generator is a java command line app that is driven by a property file (generator.properties) for configuration. An example configuration file is shown below with comments to describe entries.

#root folder for target generated project

project.root.path=C:/java/projects/kfsnodeorm

# comma-delimited source search path for ojb files etc.

source.search.paths=c:/java/projects/rice2,c:/java/projects/kfs

# db connection properties file

db.connection.config=c:/users/tucker87/dbconn.txt

# comma-delimited list of tables to ignore during generation

ignore.tables=TRV\_\*,TRVL\_\*,TEM\_\*,\*\_V

# comma-delimited list of folders to ignore during generation

ignore.folders=.git,target,webapp,test,demo,sample

# delete old generated objects when rerun

clean.model.target=true

# comma-delimited list of valid class path prefixes for source

valid.class.path.prefixes=org.kuali,edu.msu