Simple Node ORM

Node ORM for MySQL and Oracle

Rob Tucker
08/12/2019

Table of Contents

Introduction

This document describes a JavaScript-based ORM for Node.js that supports Oracle, MySQL and PostgreSQL. Simplenode ORM is designed to be imported into an existing javascript app to provide database access based on configuration files provided by the parent application as in the example ClinicalHelper app below:

```
const fs = require('fs');
const orm = require('@simplenodeorm/simplenodeorm/orm');

const appConfiguration = JSON.parse(fs.readFileSync('./appconfig.json'));
const testConfiguration = JSON.parse(fs.readFileSync('./testconfig.json'));
const customizations = require('./Customization.js');

orm.startOrm(__dirname, appConfiguration, testConfiguration, onServerStarted,
customizations);

function onServerStarted(server, logger) {
    logger.logInfo("simplenodeorm server started");

    server.get('/clinicalhelper', async function (req, res) {
        res.status(200).send("clinicalhelper call made");
    });
}
```

Simplenodeorm is available in the npm repository.

npm i @simplenodeorm/simplenodeorm

Node 10.5 or greater is required. If Oracle functionality is desired then Oracledb driver 2.2 is required. The oracle driver and client installation must be done manually as described here.

Simplenodeorm is available on github at:

https://github.com/simplenodeorm/simplenodeorm.git

Examples in this document will use the MySQL sakila demo database.

The ORM support object-based database access. OQL-like queries can be created like example below:

```
select Film o from Film
where o.filmId = :filmId
```

The framework consists of 5 main functional areas:

- Orm – the application entry point. Loads models, metadata, repositories and the database interface. Also provides REST based access to database if desired.
- Model – standard object mapping of database tables
- Repository – provides CRUD functionality to/from database using the model
- MetaData – table-to-model definitions
- Test – unit test logic

Table access and testing is provided by JavaScript objects that extend provided base objects and adhere to a standard naming convention. For example, the MySQL film table is represented by the following objects:

- Film.js – model
- FilmRepository.js – database access
- FilmMetaData.js – metadata definitions

Each repository module supports the following object-based methods:

- 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

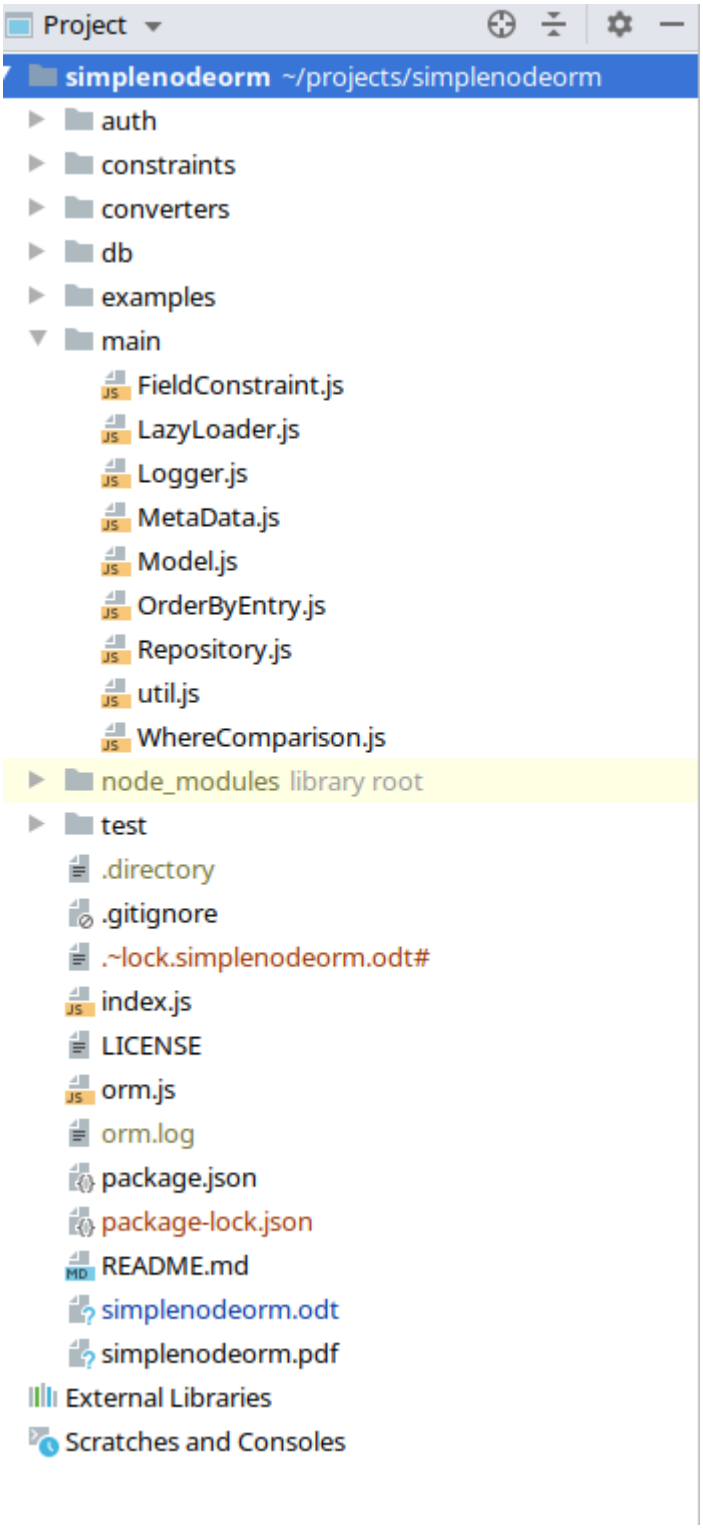
The methods above are async calls that return a Promise. Each call also has a synchronous counterpart:

- findOneSync
- findSync
- getAllSync
- countSync
- existsSync
- saveSync
- deleteSync

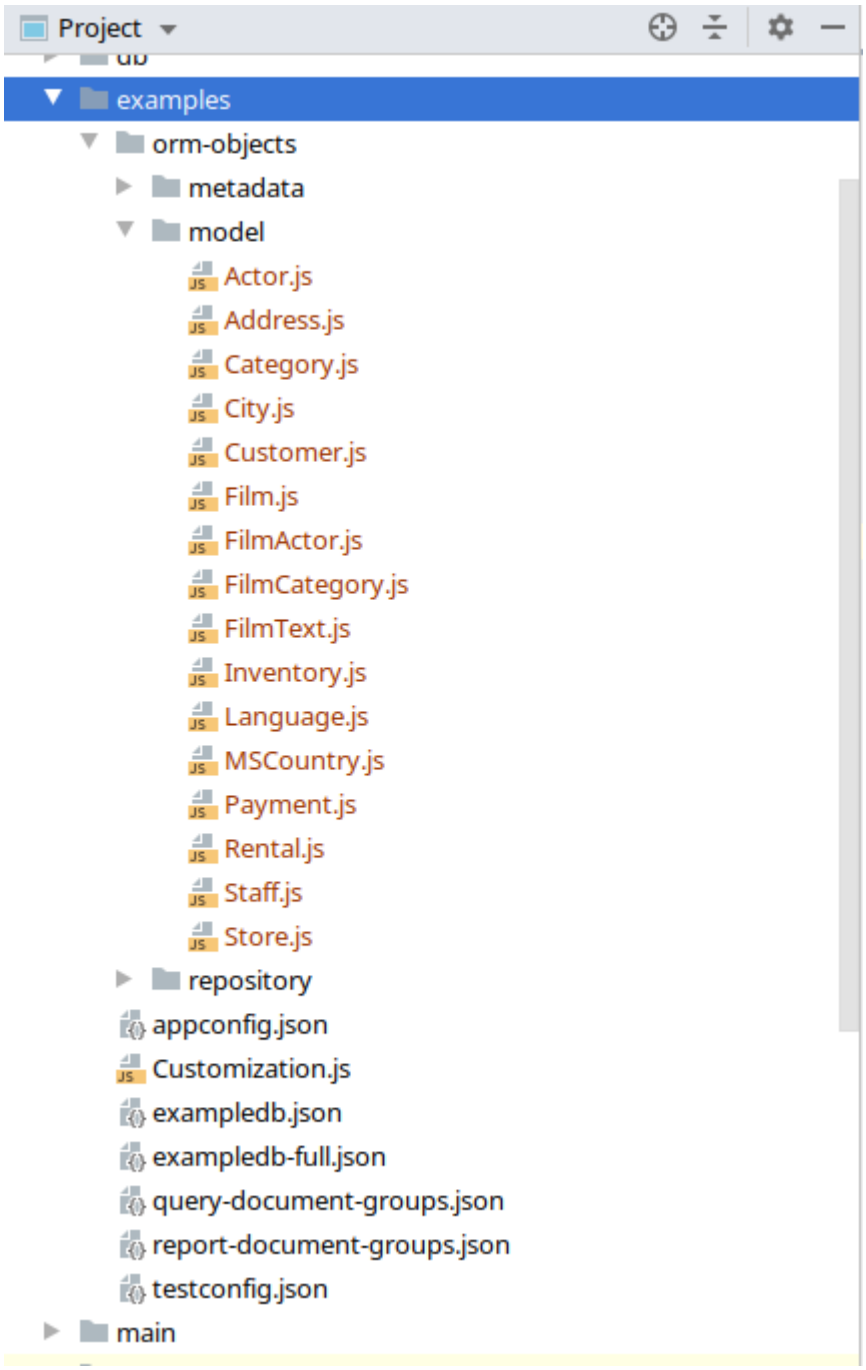
Standard SQL queries and result sets are also supported.

Project Layout

The diagram below shows the simplenodeorm project layout.


Project
simplenodeorm ~/projects/simplenodeorm
auth
constraints
converters
db
examples
main
FieldConstraint.js
LazyLoader.js
Logger.js
MetaData.js
Model.js
OrderByEntry.js
Repository.js
util.js
WhereComparison.js
node_modules library root
test
.directory
.gitignore
.~lock.simplenodeorm.odt#
index.js
LICENSE
orm.js
orm.log
package.json
package-lock.json
README.md
simplenodeorm.odt
simplenodeorm.pdf
External Libraries
Scratches and Consoles

The examples folder contains example files that would be provided by the parent application:



Project Folder Descriptions

Below you will find a short description of other key project folders.

auth

This folder contains security components for the orm. This is where you would place custom authorizers to be used for access control.

constraints

This folder contains constraint objects that can be applied when inserting and updating database tables. Be default 2 are provided – NotNullConstraint and LengthConstraint. Custom constraints can be added to a field by placing them here and adding them to the fieldConstraints map in the MetaData object for the desired field and model. If the constraintsEnabled flag on the model is set to true then constraints will be checked on inserts and updates.

converters

The converters folder contains data converters for converting data going into and out of the database. The following are provided by default:

- DecimalPrecison2
- EncryptDecrypt
- InverseYNToBoolean
- Long
- Percent
- TFToBoolen
- YNToBoolean
- ZeroOneToBoolean

To apply a converter to a field, add the converter to the MetaData field definition as shown below:

```
{
    fieldName: "someIndicator",
    type: "VARCHAR2",
    converter: "YNToBoolean",
    length: 1,
    columnName: "SOME_INDICATOR"
}
```

Custom converters can be added by placing them here.

db

Contains the database integration logic files

main

Contains the application base classes

test

Contains the default test files. The testconfig provided by the parent application should specify the locations for the insert test json files in the test/testdata folder.

Design Concepts

This ORM is designed to provide simple, configurable, efficient database access to Oracle, MySQL and PostgreSQL databases via JavaScript objects and Node. As stated in the introduction, the application consists of 5 major areas – orm, model, repository, metadata and test. Each of these areas are built with JavaScript using the node module system. A general description of each area is found below:

Orm

orm.js is the application module for simplenodeorm. The orm initiates the database interface by creating configured database pools then loads the repository and metadata modules. These modules are provided by the parent application and made available by model name via the methods below:

- orm.getRepository('modelName')
- orm.getRepository('modelName').getMetaData()

The orm provides database connections from a configured connection pool via the orm.getConnection('poolAlias') method. Multiple pools are supported. Each repository object is associated with a database pool.

Database REST access provided by the orm.js for data retrieval and access via the **Query Designer** and **Report Designer** applications.

Model

A model object is the familiar data container that maps to a database table. Each Model object will extend Model.js (code snippet shown below) to provide basic functionality:

```
"use strict";

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

class Model {
    constructor(metaData) {
        this.__model__ = metaData.objectName;
        this.metaData = metaData;
        this.modified = false;
        this.newModel = true;
        this.constraintsEnabled = false;
        this.data = new Object();
        this.initializeData();
    }

    isModified() {
        return this.modified;
    }
```

```
setModified(modified) {
    this.modified = modified;
}

isNew() {
    return this.newModel;
}

setNew(newModel) {
    this.newModel = newModel;
}

getData() {
    return this.data;
}

setData(data) {
    this.data = data;
}

getFieldValue(fieldName, ignoreLazyLoad) {
    let retval =  this.data[fieldName];

    // only lazy load when running under node
    if (!ignoreLazyLoad
        && util.isUndefined(retval)
        && util.isNodeEnv()
        && this.metaData.isLazyLoad(fieldName)) {
        if (util.isUndefined(lazyLoader)) {
            lazyLoader = require('./LazyLoader.js');
        }

        lazyLoader.lazyLoadData(this, fieldName);
    }

    return retval;
}

setFieldValue(fieldName, value) {
    // if constraints are enabled then check incoming data
    if (this.metaData && this.constraintsEnabled) {
        let constraints = this.metaData.getFieldConstraints(fieldName);

        if (util.isValidObject(constraints)) {
            for (let i = 0; i < constraints.length; ++i) {
                constraints[i].check(this.metaData.getObjectName(), fieldName, value);
            }
        }
    }

    if (!this.modified) {
        this.modified = (this.data[fieldName] !== value);
    }

    this.data[fieldName] = value;
}
```

A few remarks about the code above:

- metaData – database metadata definition associated with this object. When the object is serialized to JSON this information will be removed.
- __model__ - object name of extending class (Account, Chart etc.) with this 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 example examples/orm-objects/model/Film.js module:

```
"use strict";

const Model = require('@simplenodeorm/simplenodeorm/main/Model');

class Film extends Model {
    constructor(metaData) {
        super(metaData);
    }

    getFilmId() { return this.getFieldValue("filmId"); }
    setFilmId(value) { this.setFieldValue("filmId", value); }

    getTitle() { return this.getFieldValue("title"); }
    setTitle(value) { this.setFieldValue("title", value); }

    getDescription() { return this.getFieldValue("description"); }
    setDescription(value) { this.setFieldValue("description", value); }

    getReleaseYear() { return this.getFieldValue("releaseYear"); }
    setReleaseYear(value) { this.setFieldValue("releaseYear", 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 CRUD access is supported. Repository.js also contains the complex logic for the SQL result set to object graph conversion. Each repository object will extend the base class Repository.js a portion of which is shown below:

```
const orm = require('../orm.js');
const util = require('./util.js');
const insertSqlMap = new Map();
const updateSqlMap = new Map();
const logger = require('./Logger.js');
const dbConfig = require('../db/dbConfiguration.js');
const sleepTime = orm.appConfiguration.deasyncSleepTimeMillis || 200;
const maxDeasyncWaitTime = orm.appConfiguration.maxDeasyncWaitTime || 30000;
```

```
var deasync = require('deasync');

/**
 * 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.namedDbOperations.set(util.DELETE, this.buildDeleteNamedOperation(metaData));
        this.selectClauses = new Array();
        this.joinClauses = new Array();

        // load custom db operations in extening classes. These are object-based db operations,
        // below is an example of Account findOne():
        // select Account o from Account where o.finCoaCd = :finCoaCd
        // and o.accountNbr = :accountNbr
        // currently the 'o' alias is important because the sql
        // generator will key on 'o.'. Specify
        // with dot notations for example o.subAccounts.subAcctNbr = :subAcctNbr
        this.loadNamedDbOperations();
    }
```

A few comments about the code above:

- const orm – this is the orm 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.
- 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.
- 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 example examples/orm-objects/repository/FilmRepository.js module:

```
"use strict";

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

class FilmRepository 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 FilmRepository(metaData);
};
```

MetaData

The metadata object contains relational-to-object mapping definitions as well as the object relationship definitions. Each MetaData object will extend the base class MetaData.js a portion of which is shown below:

```
/**
 * this is the base class for defing the sql to object mapping definitions
 */
class MetaData {
    constructor(
        objectName,
        module,
        tableName,
        fields,
        oneToOneDefinitions,
        oneToManyDefinitions,
        manyToOneDefinitions) {
        this.objectName = objectName;
        this.module = module;
        this.tableName = tableName;
        this.fields = fields;
        this.oneToOneDefinitions = oneToOneDefinitions;
        this.oneToManyDefinitions = oneToManyDefinitions;
        this.manyToOneDefinitions = manyToOneDefinitions;
        this.fieldConstraints = new Map();
        this.lazyLoadFields = new Set();

        // map of column name to field definitions
        this.columnToFieldMap = new Map();

        // map of field name to field definitions
        this.fieldMap = new Map();

        // map of field name to field definitions
        this.referenceMap = new Map();

        // add some default constraints - will be disabled by default in the model object
        for (let i = 0; i < fields.length; ++i) {
            if (fields[i].lazyLoad) {
```

```
            this.lazyLoadFields.add(fields[i].fieldName);
        }

        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")));
        }
```

As you can see this is a generic class that stores the definitions. The extending class will pass in table/field definitions in constructor as JSON in the super() call. Also note that in this base class we are creating LengthConstraint and a NotNullConstraint for fields by default 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.

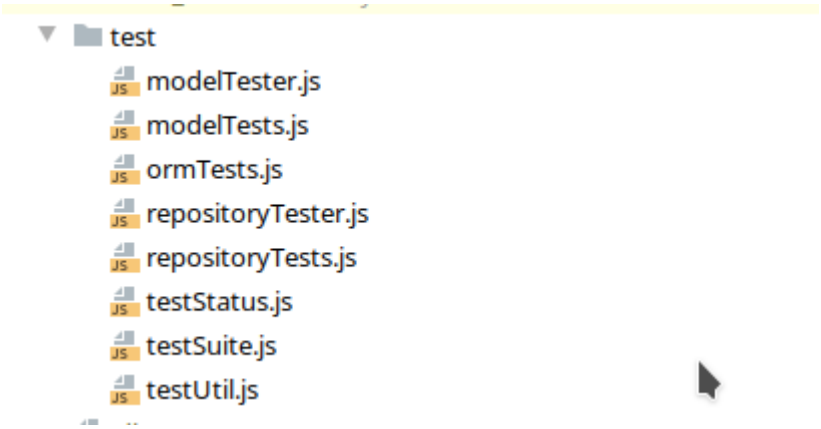
A portion of the examples/orm-objects/metadata/FilmMetaData.js module is shown below:

```
"use strict";

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

class FilmMetaData extends MetaData {
    constructor() {
        super(
            'Film', // object name,
            'model/mysql/Film.js', // relative module path,
            'film', // table name
            [ // field definitions - order is important, selected data
              // will be in this order, primary key fields should be first
                { // 0
                    fieldName: "filmId",
                    type: "SMALLINT UNSIGNED",
                    columnName: "film_id",
                    required: true,
                    primaryKey: true,
                    autoIncrementGenerator: "LAST_INSERT_ID()"
                },
                { // 1
                    fieldName: "title",
                    type: "VARCHAR",
                    length: 255,
                    columnName: "title",
                    required: true
                },
                { // 2
                    fieldName: "description",
                    type: "TEXT",
                    lob: true,
                    columnName: "description"
```

```
        .
        .
        .

        [ // one-to-one definitions
                  { // 0
                      fieldName: "language",
                      type: 1,
                      targetModelName: "Language",
                      targetModule: "../model/mysql/Language.js",
                      targetTableName: "language",
                      status: "enabled",
                      joinColumns: {
                          sourceColumns: "language_id",
                          targetColumns: "language_id"
                      }
                  },
                  { // 1
                      fieldName: "originalLanguage",
                      type: 1,
                      targetModelName: "Language",
                      targetModule: "../model/mysql/Language.js",
                      targetTableName: "language",
                      status: "enabled",
                      joinColumns: {
                          sourceColumns: "original_language_id",
                          targetColumns: "language_id"
                      }
                  }
              ],
              [], // one-to-many definitions
              []); // many-to-one definitions

    }
}

module.exports = function() {
    return new FilmMetaData();
};
```

Test

Below is the test hierarchy in the project:



If testMode in application configuration JSON provided by the parent application is set to true, then tests will be run after ORM startup. The entry point for all tests is testSuite.js.

```
"use strict";

const testUtil = require("./testUtil.js");
const ormTests = module.require("./ormTests.js");
const modelTests = module.require("./modelTests.js");
const repositoryTests = module.require("./repositoryTests.js");
const orm = require("../orm.js");

module.exports.run = async function() {
    testUtil.logInfo("running testSuite...");

    try {
        await ormTests.run(orm);
        await modelTests.run(orm);
        await repositoryTests.run(orm);
    }

    catch (e) {
        testUtil.logError('Exception in testSuite ' + e.stack);
    }

    testUtil.logInfo("testSuite complete");
};
```

The underlying test logic uses the data in an associated test database provided by the parent application in the test configuration JSON to dynamically test the CRUD capabilities of the ORM. This logic works well for the read, update and delete if each test table contains valid data. This logic does not work so well for the create/insert tests. To test create/insert functionality, insert files are created using the model name such as the example examples/testdata/Film_1.json shown below:

```
{
  "__model__":"Film",
  "modified":true,
  "newModel":true,
  "constraintsEnabled":false,
  "data":{
    "language":{
      "__model__":"Language",
      "modified":false,
      "newModel":false,
      "constraintsEnabled":false,
      "data":{
        "languageId":1,
        "name":"English",
        "lastUpdate":"2006-02-15T12:02:19.000Z"
      }
    },
    "originalLanguage":null,
    "filmId":null,
    "title":"test filem",
    "description":"test film text",
    "releaseYear":2019,
    "languageId":1,
    "rentalDuration":6,
    "rentalRate":0.99,
    "length":86,
    "replacementCost":20.99,
    "rating":"PG",
    "specialFeatures":"Deleted Scenes,Behind the Scenes",
    "lastUpdate":"2006-02-15T12:03:42.000Z"
  }
}
```


Each file contains the JSON representation of an object to be used for insert. The location for these files is provided by the parent application in the test configuration JSON. To create a file, use the results returned from a Repository.findOne() call and modify as required.

Application Configuration

There are 3 JSON files for application and testing configuration that are expected to be provided by the parent application (name is not important):

<appconfig>.json:

```
{
  "testMode" : true,
  "dbConfiguration" : "dbconfig/dbconfig.json",
  "defaultMaxJoinDepth" : 4,
```

```
  "defaultDesignTableDepth": 4,
  "createTablesIfRequired" : false,
  "apiPort" : 8443,
  "logFile" : "clinicalhelper.log",
  "logLevel" : "info",
  "deasyncSleepTimeMillis": 200,
  "maxDeasyncWaitTime" : 30000,
  "maxRowsForGetAll" : 1000,
  "authorizer": "auth/ClinicalHelperAuthorizer",
  "chartjsurl": "https://cdnjs.cloudflare.com/ajax/libs/Chart.js/2.8.0/Chart.min.js",
  "certKeyPath" : "cert/server.key",
  "certPath" : "cert/server.crt",
  "ormModuleRootPath": "orm",
  "aliases" : {
       “alias1”: “somelongname1”,
       “alias2”: “somelongname2”
       }
}
```

name	**description**
testMode	If set to true, unit tests will run after orm startup
dbConfiguration	Parent application ralative path to database configuration file that defines connection pool connection information
queryDocumentRoot	Full path tor query design document storage
reportDocumentRoot	Full path tor report design document storage
defaultMaxJoinDepth	Will determine how deep to create parent-to-child joins in generated SQL
createTablesIfRequired	If true, any model definitions that do not exist in the database will have tables, foreign keys and sequences created if required
apiPort	Port for REST access
deasyncSleepTimeMillis	Time in millis to sleep while waiting for underlying async calls to complete during a synchronous call
logFile	Full path for generated log file
logLevel	Default log level for application
maxRowsForGetAll	Limit on number of rows returned from getAll() calls
maxDeasyncWaitTime	Max time in millis to wait on async calls to complete in a sync call
aliases	By default, REST calls are of the form localhost:8443/ormapi/film/findOne?filmId=1 Where film is the desired model to access. In cases where long model names become unwieldy you can use an alias by defining it here.
authorizor	Relative path in parent to the application authorizer module

chartjsurl	This is the url for chartjs support
certKeyPath	Relative path in parent of SSL cert key
certPath	Relative path in parent of SSL certificate
ormModuleRootPath	Relative path in parent where orm definitions can be found

<poolconfig>.json

The pool configuration file is specified in the **dbConfiguration** entry of the appconfig.json. In production this would be placed in a secure location. An example fire exampledb.json can be found in the examples folder.

```
{
	"pools": [{
		"dbtype": "oracle",
		"user": "hr",
		"password": "hr",
		"connectString": "localhost/XE",
		"poolAlias": "hrdb",
		"poolMax": 20,
		"poolMin": 2,
		"poolIncrement": 5,
		"poolTimeout": 120,
		"retryCount": 3,
		"retryInterval": 500,
		"runValidationSQL": true,
		"validationSQL": "alter session set current_schema=HR"
	},
	{
		"dbtype": "mysql",
		"host": "localhost",
		"user": "root",
		"poolAlias": "sakila",
		"password": "password",
		"database": "sakila",
		"supportBigNumbers": true,
		"waitForConnections": true,
		"connectionLimit": 20
	},
```

```
        {
                "dbtype": "postgres",
                "host": "localhost",
                "database": "dvdrental",
                "user": "postgres",
                "poolAlias": "dvdrent",
                "password": "postgres",
                "port": 5432,
                "ssl": false,
                "max": 20,
                "min": 4,
                "idleTimeoutMillis": 1000,
                "connectionTimeoutMillis": 5000
        }
        ]
}
```

<testconfig>.json:

```
{
    "stopTestsOnFailure" : false,
    "testDbConfiguration" : "examples/exampledb.json",
    "maxRowsForGetAll" : 50,
    "fieldsToIgnoreForUpdate" : ["startDate", "picture", "languageId", "addressId"],
    "fieldsToIgnoreForRowToModelMatch": ["lastUpdate", "picture",
      "createDate", "paymentDate"],
    "testDataRootPath": "test/testdata"
}
```

name	**description**
stopTestsOnFailure	If set to true testing process will stop when error is encountered
testDbConfiguration	Path to test database configuration
maxRowsFoGetAll	Limit on number of rows returned from getAll() calls
fieldsToIgnoreForUpdate	The test logic looks at database tables under test and attempts to find fields that can safely be updated for update tests. By default, primary keys, foreign keys and non-nullable fields are ignored. If there are other fields that should be ignored they can be entered here.
fieldsToIgnoreForRowToModelMatch	Some fields may need to be ignored for data matching during tests. Most of these will be timestamp fields that are automatically updatred

testDataRootPath	Relative path in parent application where insert/update and querydesigner test files can be found

MetaData Details

The metadata definitions are crucial to the operation of the ORM. Database queries are generated as joins 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. A MetaData object consists of field and reference definitions. There are 3 types of reference definitions supported:

- One to Many
- One to One
- Many to One

All MetaData objects must extend the base object shown below:

```
class FilmMetaData extends MetaData {
   constructor() {
```

Required definitions are passed to the base class in the following order:

1. objectName – name of model object, for example ‘Account’
2. module – relative path to Node.js module, for example ‘model/Film.js’
3. tableName associated database table name
4. fields – JSON array of field definitions
5. oneToOneDefinitions – JSON array of one to one reference definitions (empty array if none)
6. oneToManyDefinitions - JSON array of one to many reference definitions (empty array if none)
7. manyToOneDefinitions - JSON array of many to one reference definitions (empty array if none)

Many to Many reference definitions are supported under the One to Many definitions with a link table definition. Order of the field definition array entries is important. It is required that the primary key fields appear first and in the correct order. Generated select clauses will be in the order of the field entries.

Field Definition Details

Below is a field definition from the examples/orm-objects/model/FilmMetaData.js:

```
{ // 0
   fieldName: "filmId",
   type: "SMALLINT UNSIGNED",
```

```
            columnName: "film_id",
            required: true,
            primaryKey: true,
            autoIncrementGenerator: "LAST_INSERT_ID()"
        },
```

Available field definitions are listed below:

name	required	description
fieldName	Y	Name that is associated with the database column data
type	Y	Database type name
length	N	Maximum field length for string type fields
columnName	Y	Database column name
required	N	If true, the field is not nullable
primaryKey	N	If true, then this field is part of the primary key
autoIncrementGenerator	N	In oracle, this would be the sequence name, in MySQL this is the function that gets the last generated id
defaultValue	N	If defined and the field is null, the defaultValue will be used in inserts and updates
converter	N	If defined, the named converter will be used to convert values when retrieving/saving values to/from the database.
lob	N	If true, this indicates a LOB field
lazyLoad	N	If true, lazy load will be performed when getField() called
decimalDigits	N	Precision for numeric fields

Join Strategy

To implement database references the ORM builds SQL joins to pull data as a result set then populates the object graph from this result set. For efficiency reasons the following join strategy is used:

1. One-to-Many joins are created to the depth specified in the appconfig.json value

   “defaultMaxJoinDepth”
2. One-to-One and Many-to-One joins are only created for the top-level object of the hierarchy.
3. All other defined references will be populated via lazy load if/when the associated Model.getField() method is called.

Reference Field Definitions

Join definitions define parent-child relationships between tables. 3 types are supported one-to-one and one-to-many and many-to-one. Many-to-many joins are

supported via a one-to-many definition with an associated join table defined. Below are some example definitions:

One-to-One Definition

```
{ // 0
    fieldName: "language",
    type: 1,
    targetModelName: "Language",
    targetModule: "model/Language.js",
    targetTableName: "language",
    status: "enabled",
    joinColumns: {
        sourceColumns: "language_id",
        targetColumns: "language_id"
    }
},
```

One-to-Many Definition

```
    { // 0
        fieldName: "films",
        type: 2,
        targetModelName: "Film",
        targetModule: "model/Film.js",
        targetTableName: "film",
        status: "enabled",
        joinColumns: {
            sourceColumns: "film_id",
            targetColumns: "film_id"
        }
    }
```

One-to-Many (with many-to-many configuration)

```
        { // 0  an example
         fieldName: "prerequisites",
         type: 3
         targetModelName: "TermSpecification",
         targetModule: "model/TermSpecification.js",
         targetTableName: "TERM_SPEC_T",
         joinTableName: "TERM_RSLVR_INPUT_SPEC_T",
         status: "enabled",
         joinColumns : {
              sourceColumns : "TERM_RSLVR_ID",
              targetColumns : "TERM_RSLVR_ID",
              inverseSourceColumns : "TERM_SPEC_ID",
              inverseTargetColumns : "TERM_SPEC_ID"
          }
      },
```

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

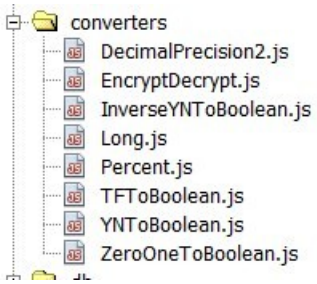
name	required	description
fieldname	Y	The field name used for access in the Model object
targetModelName	Y	Model name associated with the linked database table
targetModel	Y	Relative path to the node.js module for the target model
targetTableName	Y	Linked database table name
type	Y	1, 2 or 3 where 1=one-to-one, 2=one-to-many and 3=many-to-one
cascadeUpdate	N	If true, parent update/inserts will cascade down to children
cascadeDelete	N	If true, parent deletes will cascade down to children
Status	Y	Enabled or disabled. You can turn off defined relationships if desired
Required	N	If true, an inner join will be created for this reference, otherwise an outer join will be created.

joinColumns	Y	Comma delimited list of source table to target table columns
		joinColumns : {
		sourceColumns : "ID1,ID2",
		targetColumns : "TID1,TID2"
		}

joinTableName	N	Used to support many-to-many references, define the linking table. If this is defined, then the join columns must contain the associated inverse column definitions, for example:
		joinColumns : {
		sourceColumns : "TERM_RSLVR_ID",
		targetColumns : "TERM_RSLVR_ID",
		inverseSourceColumns : "TERM_SPEC_ID",
		inverseTargetColumns : "TERM_SPEC_ID"
		}

Converters

The converter on the JSON field definition can specify an object to use to convert data to/from the database. You can find some default converter under the converters folder.



The YNToBoolean converter is shown below. All converters should follow this pattern where “field” is Field object, “value” is the value to be converted and “fromDb” is a Boolean indication that the value is coming from the database.

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

module.exports = function(field, value, fromDb) {
    let retval = value;

    if (util.isValidObject(value)) {
        if (fromDb) {
            retval = (value === 'Y');
        } else {
            if (value) {
                retval = 'Y';
            } else {
                retval = 'N';
            }
        }

    }

    return retval;
};
```

Repository Details

The bulk of the work in the ORM goes on in the Repository.js base class. All custom repository objects must extend Repository:

```
"use strict";

const poolAlias = 'sakila';
const Repository = require('@simplenodeorm/simplenodeorm/main/Repository');

class FilmRepository 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 FilmRepository(metaData);
};
```

When running queries, SQL is generated as one select statement with joins to related tables as defined by the one-to-one, one-to-many and many-to-one 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 (root) class always designated with the alias “t0” and will be the only table to define joins on one-to-one and many-to-one relationships. Other relationship definitions will be executed via lazy-load logic. 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 defaultMaxJoinDepth will be used.
Below is an example of the SQL generated findOne with defaultMaxJoinDepth = 4.

```
select
  t0.film_id as t0_film_id,
  t0.title as t0_title,
  t0.description as t0_description,
  t0.release_year as t0_release_year,
  t0.language_id as t0_language_id,
  t0.original_language_id as t0_original_language_id,
  t0.rental_duration as t0_rental_duration,
  t0.rental_rate as t0_rental_rate,
  t0.length as t0_length,
  t0.replacement_cost as t0_replacement_cost,
  t0.rating as t0_rating,
  t0.special_features as t0_special_features,
  t0.last_update as t0_last_update,
  t0_t5_0.language_id as t0_t5_0_language_id,
  t0_t5_0.name as t0_t5_0_name,
  t0_t5_0.last_update as t0_t5_0_last_update,
  t0_t6_0.language_id as t0_t6_0_language_id,
  t0_t6_0.name as t0_t6_0_name,
  t0_t6_0.last_update as t0_t6_0_last_update
from
  film t0
  left outer join language t0_t5_0 on (
    t0_t5_0.language_id = t0.language_id
  )
  left outer join language t0_t6_0 on (
    t0_t6_0.language_id = t0.original_language_id
  )
where
  t0.film_id = ?
```

And here are the JSON results from a findOne() call the call above:


```
 {
   "__model__":"Film",
   "modified":false,
   "newModel":false,
   "constraintsEnabled":false,
   "data":{
      "language":{
         "__model__":"Language",
         "modified":false,
         "newModel":false,
         "constraintsEnabled":false,
         "data":{
            "languageId":1,
            "name":"English",
            "lastUpdate":"2006-02-15T12:02:19.000Z"
         }
      },
      "originalLanguage":null,
      "filmId":1,
      "title":"ACADEMY DINOSAUR",
```

```
      "description":"A Epic Drama of a Feminist And a Mad Scientist who must Battle a Teacher in
The Canadian Rockies",
      "releaseYear":2006,
      "languageId":1,
      "rentalDuration":6,
      "rentalRate":0.99,
      "length":86,
      "replacementCost":20.99,
      "rating":"PG",
      "specialFeatures":"Deleted Scenes,Behind the Scenes",
      "lastUpdate":"2006-02-15T12:03:42.000Z"
   }
}
```


Object Based Database Operations

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

```
select Employee o
from Employee
where o.employeeId = 1
order by o.lastName,
o.firstName
```

The “o.” prefix shown above is significant and is required in the OQL. Field names can be specified using dot notation to designate child objects, for example:

```
select Film o
from Film
where o.title = ‘someTitle’ and o.releaseYear = 2006
order by o.title
```

Predefined SQL operations can be added and used by the repository by overriding the
Repository.loadNamedDbOperations() method and adding the custom operations as shown below:

```
loadNamedDbOperations() {
getNamedDbOperations().add(“findByTitle”,
    “select File o from Film where o.title
                = :title”);  }
```

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

```
let params = new Array();
params.push(‘myLastName) let
repo =  orm.getRepository(‘Film’);
let result = await repo.executeNamedDbOperation(‘findByTitle’, 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_information> } 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 object (by id) 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 there is no where clause the table row count is returned.

async count(whereComparisons, options)

- whereComparisons – optional 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_information>} 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 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 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/nodeoracledb/blob/master/doc/api.md#executeoptions

- 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. 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 construct SQL based on the maxDefaultJoinDepth value from the appconfig.json. If you want to override this setting for an individual 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 results will be returned in result.updatedValues.

WhereComparison.js

To create a complex where clause the Where Comparison module is used A WhereComparison 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:

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

/**
 * this object defines one comparison entry for a where clause.
 */
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;
            this.comparisonValue = '';
        }

        if (util.IN === comparisonOperator) {
            this.useBindParams = false;
        }
    }

    getFieldName() {
        return this.fieldName;
    }
```

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;
    }
}

module.exports.OrderByEntry =
OrderByEntry;

module.exports= function(fieldName,
descending) {
    if (util.isUndefined(descending))
{
        descending = false;
    }
    return new OrderByEntry(fieldName,
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 Employee.job.jobTitle
- descending – true/false – defaults to false

Database Configuration

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. Database connection parameters are expected to be found outside the application in a JSON file as described in the Application Configuration section of this document. An example of the database configuration file used to connect to HR Demo database is shown below:

```
{
	"pools": [
	{
		"dbtype": "mysql",
		"host": "localhost",
		"user": "root",
		"poolAlias": "sakila",
		"password": "password",
		"database": "sakila",
		"supportBigNumbers": true,
		"waitForConnections": true,
		"connectionLimit": 20
	}
	]
}
```

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

```
"use strict";


const util = require("../main/util.js");
const fs = require('fs');
const logger = require('../main/Logger.js');


// try the various supported databases - ignore errors, assume no driver present
let oracledb;
try {
    oracledb = require('oracledb');
} catch(e) {}
```

```
let mysqldb;
try {
    mysqldb = require('promise-mysql');
} catch (e) {}



let postgresdb;
try {
    postgresdb = require('pg-pool');
} catch (e) {}


module.exports = function(poolCreatedEmitter, appConfiguration,
       testConfiguration, dbTypeMap) {
    if (appConfiguration.testMode) {
        initPool(testConfiguration.testDbConfiguration, poolCreatedEmitter, dbTypeMap);
    } else {
        initPool(appConfiguration.dbConfiguration, poolCreatedEmitter, dbTypeMap);
    }
};


async function initPool(securityPath, poolCreatedEmitter, dbTypeMap) {
    logger.logInfo("creating connection pools...");

    // read db connection info
    let pdefs = JSON.parse(fs.readFileSync(securityPath));
    let haveOracle = false;
    for (let i = 0; i < pdefs.pools.length; ++i) {
        let pool;
        switch(pdefs.pools[i].dbtype) {
            case util.ORACLE:
                pool = await oracledb.createPool(pdefs.pools[i]);
                haveOracle = true;
                break;
            case util.MYSQL:
                pool = await mysqldb.createPool(pdefs.pools[i]);
                break;
            case util.POSTGRES:
                pool = new postgresdb (pdefs.pools[i]);
                break;
        }



        if (pool) {
            logger.logInfo("    " + pdefs.pools[i].poolAlias
                    + " connection pool created");
            dbTypeMap.set(pdefs.pools[i].poolAlias, pdefs.pools[i].dbtype);
            dbTypeMap.set(pdefs.pools[i].poolAlias + '.pool', pool);
        } else {
```

```
            logger.logWarning('invalid dbtype: ' + pdefs.pools[i].dbtype)
        }
    }
```

Getting Started

Below are the steps to use simplenodeorm in your project:

1. Install simplenodeorm version 1.4.3 or higher using npm:
   npm install @simplenodeorm/simplenodeorm
2. In your application require orm:
   ```
   const orm = require('@simplenodeorm/simplenodeorm');
   ```
3. Load you configuration and call the startOrm function:
   ```
   const appConfiguration = JSON.parse(fs.readFileSync('./appconfig.json'));
   const testConfiguration = JSON.parse(fs.readFileSync('./testconfig.json'));
   const customizations = require('./Customization.js');


   orm.startOrm(__dirname, appConfiguration, testConfiguration,
       onServerStarted, customizations);

function onServerStarted(server, logger) {
logger.logInfo("simplenodeorm server started");

server.get('/clinicalhelper', async function (req, res) {
res.status(200).send("clinicalhelper call made");
});
}
```

It is expected that the required ORM object have been created and are located in the location specified by <appconfig>.ormModuleRootPath. The callback function (“onServerStart” above) will be called with the express server instance and the server logger. You can use this server instance to add you own customized http handlers.

Some Example Code

The code below shows some example calls using a model with name “Employee”:

findOne

```
let repo = orm.getRepositry(‘Film’);
let params = new Array();
params.push(1);
```

```
let res = repo.findOne(params);
if (util.isDefined(res.error)) {
     // handle error
} else {
    return res.result;
}
```

find

```
let repo = orm.getRepositry('Film');
let whereComparisons = new Array();
whereComparisons.push(require('../main/WhereComparison.js')
    ('releaseYear', 2006, util.EQUAL_TO));
let orderByEntries = new Array();
 orderByEntries.push(require('../main/OrderByEntry.js')('title');

let res = repo.find(whereComparions, orderByEntries); if
(util.isDefined(res.error)) {
     // handle error
} else {
    return res.result;
}
```

count

```
let repo = orm.getRepositry('Film');
let whereComparisons = new Array();
whereComparisons.push(require('../main/WhereComparison.js')

    ('releaseYear', 2006, util.EQUAL_TO));

let res = repo.count(whereComparions);
if (util.isDefined(res.error)) {
    // handle error
} else {
   return res.result;
}
```

exists

```
let repo = orm.getRepositry('Film');

let model =
    orm.newModelInstance(orm.getMetaData('Film');
model.setReleaseYear(1995);
let res = repo.exists(model);
if (util.isDefined(res.error)) {
    // handle error
} else {
    return res.result;
}
```

getAll

```
let res =
    orm.getRepositry('Film').getAll();
 if (util.isDefined(res.error))
{
     // handle error
} else {
    return res.result;
```

```
}
```

save

```
let repo = orm.getRepositry('Film);
let conn = orm.getConnection(repo.getAlias());
let params = new Array();
params.push(1);
let model = repo.findOne(params, {conn: conn});
model.setTitle('new title');
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('Film');
let conn = orm.getConnection(repo.getAlias());
let params = new Array();
params.push(1);
let model = repo.findOne(params, {conn: conn});
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.getRepository('Employee');
repo.executeSqlQuery('select releaseYear from Film where title = 'some title' order by 1');
if (util.isDefined(res.error)) {
     // handle error
} else {
    return res.result;
}
```

executeSql

```
let repo = orm.getRepository('Film');
let conn = orm.getConnection(repo.getAlias());
repo.executeSql('update Film set title = \'some title\' where filmId = 100,
[], {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 within the application using the Node express package. In addition to the basic ORM functionality, REST access is provided for the Query and Report Designer applications. The API supports the base repository calls – findOne, find, exists, count, save and delete. The URL used is something like:

http://localhost:<restPort>/ormapi/<model-name- lowercase>/<operation>?param1=MS&param2=RA123456

for example:
http://localhost:8888/ormapi/film/findOne?filmId=100

where “film” 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.