|  |  |
| --- | --- |
| **Jira ID:** | [**CHE-24**](http://jira.chenoainc.com:8080/browse/CHE-24) |
| **Application Name:** | **Rapid-Start-NodeJS** |
| **Author:** | Arinjay A. Jain/Ishan P. |
| **Date Created:** | **8/6/2018** |

**Revision History**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Date** | **Version No.** | **Summary of Changes** | **Revised By** | **Approved By/Date** |
| 08/08/2018 | 0.1 | Draft | Anil Jain | Kamliesh N. |
|  |  |  |  |  |

Contents

[1 Introduction 4](#_Toc521936920)

[1.1 GitHub URL 4](#_Toc521936921)

[1.2 Technology Specifications 4](#_Toc521936922)

[2 Functionalities 4](#_Toc521936923)

[2.1 Basic framework 4](#_Toc521936926)

[2.2 Log4j 5](#_Toc521936927)

[2.3 Caching 5](#_Toc521936928)

[2.4 Insert records in user\_masters table 5](#_Toc521936929)

[2.5 Insert records in role\_masters table 6](#_Toc521936930)

[2.6 Fetch all users from user\_masters table 6](#_Toc521936931)

[2.7 User Batch Upload 7](#_Toc521936932)

[2.7.1 Insert into batch\_master table 8](#_Toc521936933)

[2.7.2 Insert into batch\_execution status table 8](#_Toc521936934)

[2.7.3 Insert into batch\_exception table 8](#_Toc521936935)

[3 Multi-Tenancy 9](#_Toc521936936)

[3.1 GitHub URL 9](#_Toc521936937)

[3.2 Basic Framework 9](#_Toc521936938)

[3.3 Log4j 10](#_Toc521936939)

[3.4 Caching 10](#_Toc521936940)

[3.5 Insert records in user\_masters table 11](#_Toc521936941)

[3.6 Insert records in role\_masters table 12](#_Toc521936942)

[3.7 Fetch all users from user\_masters table 13](#_Toc521936943)

[3.8 User Batch Upload 13](#_Toc521936944)

[3.8.1 Insert into batch\_master table 14](#_Toc521936945)

[3.8.2 Insert into batch\_execution status table 15](#_Toc521936946)

[3.8.3 Insert into batch\_exception table 15](#_Toc521936947)

[3.9 Authentication and Authorization 16](#_Toc521936948)

[3.10 Unit Testing and Coverage 17](#_Toc521936949)

# Introduction

The following document illustrates the various functionalities available as part of rapid framework for Nodejs.

* 1. GitHub URL

<https://github.com/aaajain/rapid-start-coe.git>

master branch – This contains services which do not support multi tenancy.

mt branch – This contains services which support multi tenancy.

* 1. Technology Specifications

NodeJS – 8.11.3

Mongo DB – 4.0.1

Mongoose (ORM) – 5.2.3

Bcrypt - 3.0.0

mongoose-multitenancy - 0.1.19

Jsonwebtokens – 8.3.0

Mocha – 5.1.0

Sinon – 4.5.0

Sinon-mongoose – 2.2.1

Nyc – 11.7.1

# Functionalities

2. 1. Basic framework

The basic framework uses express framework for web app. The framework uses settings.js file for environment config.

var settings = {

"mongo": {

"url":"mongodb://localhost:27017/coe"

},

"contextRoot”: '/services/',

"logLevel”: 'DEBUG',

"appPort": "9090",

"timeOut": 500,

}

module.exports =settings;

To start the server, we need to type node app.js

The server starts on the configured port. When the application starts, it invokes mongo-connect file and creates a mongo db connection which is used in all the subsequent requests. Also, the application creates an admin role and an admin user. So, the server startup creates two documents viz role\_master and user\_master.

* 1. Log4j

To configure the log4j spenders we need to use config/logger.js. If we want to enable logging in any new file, we need to import the logger.js as follows-

var logger = require('../config/logger.js').getLogger('query-utils');

logger.debug('roles updated’);

This shall log the above line in logs/server.log (this is configured in logger.js)

[2018-07-23 21:10:08.864] [DEBUG] query-utils - admin updated

* 1. Caching

The master tables shall be cached during the server start up. Sample tables that framework caches are role\_masters and tenant master. Other master tables can be cached in similar way. The cached tables are available as public objects.

queryUtils.roleMasterData

queryUtils.tenantData

* 1. Insert records in user\_masters table

The user\_masters table consists of the following columns: -

username

email

password

role

URL: - <http://host:port/services/createUser>

Method: - POST

Records will be inserted into this table using the JSON format. The sample JSON is as follows: -

{

"username”: "DevTest",

"email”: "test@test.com",

"password”: "test",

"role\_name”: "Intern",

"logged\_in\_user”: "xyz"

}

The role\_name column references the role\_id column of the role\_masters table and the corresponding role\_id is fetched, for the specified role\_name, which is then inserted into the user\_masters table under the “role” column. The logged\_in\_user is simply to verify whether the user has access to perform any operation on the database or not. If the user does not have access then the record will not be inserted into the user\_masters table. If the email, password or role\_name fields are altered for a particular username, then the entry for that username will be updated in the database. No new entry will be inserted into the database for this scenario. The password will be inserted into the database in an encrypted format. This has been done using the “bcrypt” module. Before the password is inserted into the database, it will be encrypted as follows: -

var salt = bcrypt.genSaltSync(constants.SALT\_ROUNDS);

var hash = bcrypt.hashSync(password, salt);

Here, the SALT\_ROUNDS will be read from a constants file. In order to utilize this encryption functionality, we should have the “bcrypt” module. If it is not present, then we can do the following: - npm install bcrypt.

* 1. Insert records in role\_masters table

The role\_masters table consists of the following columns: -

Role\_name

Permissions

URL: - <http://host:port/services/insertRole>

Method: - POST

Records will be inserted into this table using the JSON format. The sample JSON is as follows: -

{

"role\_name”: "Test",

"permissions”: [ "create","view","modify","create"],

"logged\_in\_user”: "admin"

}

The logged\_in\_user checks whether the user has access or not. If the user does not have access then the record will not be inserted into the database. Also, for the same role\_name, if the permissions are altered, then the record will be updated in the database. For this scenario, no new entry will be inserted into the database.

* 1. Fetch all users from user\_masters table

This service will fetch all the users from the user\_masters table.

URL: - <http://host:port/services/getAllUsers>

Method: - GET

Input Parameters: - logged\_in\_user

If the logged\_in\_user has access, then the records will be fetched from the database. If the logged\_in\_user does not have access then no records will be fetched.

* 1. User Batch Upload

This module will insert a batch of users into the user\_masters table in the database. Records will be inserted into this table using the JSON format. These records will be read from the uploadUserBatch.json file. After the json file is read, an entry will be made in the batch\_master table for the user batch upload. Simultaneously, an entry will also be made in the batch\_execution\_status table, for each user in the batch file, with the is\_success column set as “Y”. If there is an error/exception while inserting the users from the user batch file, the is\_success column would be set as “N” for that particular user and an entry would be made in batch\_exception table for that user. The sample JSON is as follows: -

{

"username”: "John",

"email”: "john@john.com",

"password”: "john",

"role\_name”: "Architect"

},

{

"username”: "Alex",

"email”: "alex@alex.com",

"password”: "alex",

"role\_name”: "Lead"

},

{

"username”: "suraj",

"email”: "suraj@suraj.com",

"password”: "suraj",

"role\_name”: "Intern"

}

The password will be inserted into the database in an encrypted format. This has been done using the “bcrypt” module. Before the password is inserted into the database, it will be encrypted as follows: -

var salt = bcrypt.genSaltSync(constants.SALT\_ROUNDS);

var hash = bcrypt.hashSync(password, salt);

Here, the SALT\_ROUNDS will be read from a constants file. In order to utilize this encryption functionality, we should have the “bcrypt” module. If it is not present, then we can do the following: - npm install bcrypt.

The role\_name column references the role\_id column of the role\_masters table and the corresponding role\_id is fetched, for the specified role\_name, which is then inserted into the user\_masters table under the “role” column.

To execute this module, do the following: - node uploadUserBatch.js

### Insert into batch\_master table

When we will execute the following command: - node uploadUserBatch.js, an entry for the user batch upload will be made in the batch\_master table. The batch\_master table consists of the following columns: -

batch\_size

batch\_name

origin

If the batch file contains 10 users then the batch\_size will be entered as 10 in the database. The batch\_name can be any name that we want to give our batch upload. Origin indicates from where the request originated. In this case, as the request originated from node js, we have set the origin as “node”. The batch\_id that will be generated for the user batch upload, will also be referenced across the batch\_execution\_status and the batch\_exception tables.

### Insert into batch\_execution status table

When we execute the command: - node uploadUserBatch.js, an entry is inserted into the batch\_execution\_status table, for the user batch upload, with the is\_success column set as “Y”. The batch\_execution\_status table consists of the following columns: -

parameter\_name

parameter\_value

is\_success

batch\_id

The parameter\_name has been set as username which is being read from a constants file. Parameter\_value is the actual username (that has been inserted under the username column of the user\_masters table in the database). Is\_success column is set to “Y”, for each user, when that user record has been successfully inserted into the user\_masters table. In case of an exception, the is\_success column is set as “N”, for that particular user record, and an entry is inserted into the batch\_exception table for that user record. The batch\_id column references the batch\_id of the batch\_master table.

### Insert into batch\_exception table

In case an exception occurs, for a particular user record, during the user batch upload, then an entry will be inserted into the batch\_exception table for that user record. The batch\_exception table consists of the following columns: -

batch\_id

parameter\_name

parameter\_value

exception\_reason

exception\_desc

origin

The batch\_id column references the batch\_id of the batch\_master table. The parameter\_name has been set as username which is being read from a constants file. Parameter\_value is the actual username (that has been inserted under the username column of the user\_masters table in the database). Exception\_reason and exception\_desc columns have been set to null for now. Origin indicates from where the request originated. In this case, as the request originated from node js, we have set the origin as “node”. The records will be inserted into this table only when the exception method has been called, that is when an exception occurs during the insertion of records in the user\_masters table. If a particular record throws an exception, then an entry for that record will be made in the batch\_exception table, and then the user batch upload process will continue to insert the subsequent user records in the user\_masters table.

# Multi-Tenancy

3.1 GitHub URL

<https://github.com/aaajain/rapid-start-coe.git>

mt branch – This contains services which support multi tenancy.

3.2 Basic Framework

The basic framework uses express framework for web app. The framework uses settings.js file for environment config.

var settings = {

"mongo": {

"url":" mongodb://localhost:27017/MT"

},

"contextRoot”: '/services/',

"logLevel”: 'DEBUG',

"appPort": "9090",

"timeOut": 500,

}

module.exports =settings;

To start the server, we need to type: - node app.js

The server starts on the configured port. When the application starts, it invokes mongo-connect file and creates a mongo db connection which is used in all the subsequent requests. Also, the application creates an admin role and an admin user, for the respective tenant’s tables. So, the server startup creates five documents: -

Tenant\_1.role\_masters

Tenant\_1.user\_masters

Tenant\_2.role\_masters

Tenant\_2.user\_masters

tenant\_master

The tenant\_master table will consist of two columns, tenant\_name and status. The tenant\_master table will have two records under the tenant\_name column: -

Tenant\_1

Tenant\_2

Both these records will have their status as “active”. These tenant records are read from the uploadTenants.json file.

## Log4j

To configure the log4j spenders we need to use config/logger.js. If we want to enable logging in any new file, we need to import the logger.js as follows-

var logger = require('../config/logger.js').getLogger('query-utils');

logger.debug('roles updated’);

This shall log the above line in logs/server.log (this is configured in logger.js)

When we do node app.js, the following logs appear: -

[2018-08-10 13:01:10.251] [DEBUG] query-utils - record inserted for tenant upload

[2018-08-10 13:01:10.269] [DEBUG] query-utils - role updated

[2018-08-10 13:01:10.286] [DEBUG] query-utils - admin updated

[2018-08-10 13:01:10.288] [DEBUG] query-utils - role updated

[2018-08-10 13:01:10.293] [DEBUG] query-utils - admin updated

The first log is generated when the tenant\_master table is created and the records from uploadTenants.json file have been inserted into the tenant\_master table. The subsequent logs are generated when the other four tables (as mentioned above) are created and records for an admin role and an admin user have been inserted into these four tables respectively.

## Caching

The master tables shall be cached during the server start up. Sample tables that framework caches are role\_masters and tenant master. Other master tables can be cached in similar way. The cached tables are available as public objects.

queryUtils.roleMasterData

queryUtils.tenantData

## Insert records in user\_masters table

There are 2 user\_masters tables – Tenant\_1.user\_masters and Tenant\_2.user\_masters. They consist of the following columns: -

username

email

password

role

URL: - http://host:port/[services](http://host:port/services/createUser)/createUser

Method: - POST

Records will be inserted into this table using the JSON format. The sample JSON is as follows: -

{

"username”: "FinTest",

"email”: "ftest@ftest.com",

"password”: "ftest",

"logged\_in\_user”: "admin",

"role\_name”: "Intern",

"tenant\_name”: "Tenant\_1"

}

The role\_name column references the role\_id column of the Tenant\_1.role\_masters table and the corresponding role\_id is fetched, for the specified role\_name, which is then inserted into the Tenant\_1.user\_masters table under the “role” column. The logged\_in\_user is simply to verify whether the user has access to perform any operation on the database or not. If the user does not have access then the record will not be inserted into the Tenant\_1.user\_masters table. If the email, password or role\_name fields are altered for a particular username, then the entry for that username will be updated in the database. No new entry will be inserted into the database for this scenario. The password will be inserted into the database in an encrypted format. This has been done using the “bcrypt” module. Before the password is inserted into the database, it will be encrypted as follows: -

var salt = bcrypt.genSaltSync(constants.SALT\_ROUNDS);

var hash = bcrypt.hashSync(password, salt);

Here, the SALT\_ROUNDS will be read from a constants file. In order to utilize this encryption functionality, we should have the “bcrypt” module. If it is not present, then we can do the following: - npm install bcrypt

The tenant\_name field specifies which tenant table should the record be inserted into. For example, in this case, as the tenant\_name has been specified as “Tenant\_1”, hence, the record will be inserted into the Tenant\_1.user\_masters table. If the tenant does not exist, then the record will not be inserted into the database. In this scenario, as we have only two tenants- “Tenant\_1” and “Tenant\_2” in our system and, if we specify the tenant\_name as “Tenant\_3”, then we will be shown the log – “tenant does not exist” and the record will not be inserted into the table. Moreover, before we execute this service, we will have to execute the login service (as explained under Authentication and Authorisation section) in order to generate the auth token. After the token is generated, we can use this token to execute the service to insert records in the specified tenant’s user\_masters table. Additionally, separate tokens will have to be generated for “Tenant\_1” and “Tenant\_2” in order to insert the user records in their respective tenant tables.

3.6 Insert records in role\_masters table

There are 2 role\_masters tables – Tenant\_1.role\_masters and Tenant\_2.role\_masters. They consist of the following columns: -

Role\_name

Permissions

URL: - <http://host:port/services/insertRole>

Method: - POST

Records will be inserted into this table using the JSON format. The sample JSON is as follows: -

{

"role\_name”: "FinTest",

"permissions”: [ "create”, “view"],

"logged\_in\_user”: "admin",

"tenant\_name”: "Tenant\_2"

}

The logged\_in\_user checks whether the user has access or not. If the user does not have access then the record will not be inserted into the database. Also, for the same role\_name, if the permissions are altered, then the record will be updated in the database. For this scenario, no new entry will be inserted into the database. The tenant\_name column indicates which tenant table should the record be inserted into. For example, in this case, since the tenant\_name is “Tenant\_2”, therefore the record will be inserted into the Tenant\_2.role\_masters table. In this scenario, as we have only two tenants- “Tenant\_1” and “Tenant\_2” in our system and, if we specify the tenant\_name as “Tenant\_3”, then we will be shown the log – “tenant does not exist” and the record will not be inserted into the table. Moreover, before we execute this service, we will have to execute the login service (as explained under Authentication and Authorisation section) in order to generate the auth token. After the token is generated, we can use this token to execute the service to insert records in the specified tenant’s role\_masters table. Additionally, separate tokens will have to be generated for “Tenant\_1” and “Tenant\_2” in order to insert the role records in their respective tenant tables.

3.7 Fetch all users from user\_masters table

This service will fetch all the users from the user\_masters table.

URL: - <http://host:port/services/getAllUsers>

Method: - GET

Input Parameters: - logged\_in\_user, tenant\_name

If the logged\_in\_user has access and the tenant\_name exists in the system (in our case either Tenant\_1 or Tenant\_2) then the records will be fetched from the specified tenant table, in the database. If the logged\_in\_user does not have access then no records will be fetched. Also, if the tenant does not exist, then no records will be fetched from the database. For example, if the tenant\_name has been specified as “Tenant\_1”, then all the records from Tenant\_1.user\_masters table will be fetched from the database. If the tenant\_name is specified as “Tenant\_3” then no records will be fetched as that tenant does not exist in our system. Moreover, before we execute this service, we will have to execute the login service (as explained under Authentication and Authorisation section) in order to generate the auth token. After the token is generated, we can use this token to execute the service to fetch all the records from the specified tenant’s user\_masters table. Additionally, separate tokens will have to be generated for “Tenant\_1” and “Tenant\_2” in order to fetch the records from their respective tenant tables.

3.8 User Batch Upload

This module will insert a batch of users into the user\_masters table, depending upon the tenant\_name, in the database. Records will be inserted into this table using the JSON format. These records will be read from the uploadUserBatch.json file. After the json file is read, an entry will be made in the batch\_master table for the user batch upload. Simultaneously, an entry will also be made in the batch\_execution\_status table, for each user in the batch file, with the is\_success column set as “Y”. If there is an error/exception while inserting the users from the user batch file, the is\_success column would be set as “N” for that particular user and an entry would be made in batch\_exception table for that user. The sample JSON is as follows: -

{

"username”: "Nitendra",

"email”: "nitendra@nitedra.com",

"password”: "nitendra",

"role\_name”: "Lead",

"tenant\_name”: "Tenant\_2"

},

{

"username”: "Ashraf",

"email”: "ashraf@ashraf.com",

"password”: "ashraf",

"role\_name”: "admin",

"tenant\_name”: "Tenant\_1"

},

{

"username”: "Jyoti",

"email”: "jyoti@jyoti.com",

"password”: "jyoti",

"role\_name”: "admin",

"tenant\_name”: "Tenant\_2"

}

The password will be inserted into the database in an encrypted format. This has been done using the “bcrypt” module. Before the password is inserted into the database, it will be encrypted as follows: -

var salt = bcrypt.genSaltSync(constants.SALT\_ROUNDS);

var hash = bcrypt.hashSync(password, salt);

Here, the SALT\_ROUNDS will be read from a constants file. In order to utilize this encryption functionality, we should have the “bcrypt” module. If it is not present, then we can do the following: - npm install bcrypt.

The role\_name column references the role\_id column of the role\_masters table and the corresponding role\_id is fetched, for the specified role\_name, which is then inserted into the user\_masters table under the “role” column.

To execute this module, do the following: - node uploadUserBatch.js

The tenant\_name field specifies which tenant table should the record be inserted into. For example, in this case, if the tenant\_name has been specified as “Tenant\_1”, then, the record will be inserted into the Tenant\_1.user\_masters table and if the tenant\_name has been specified as “Tenant\_2” then the record will be inserted into the Tenant\_2.user\_masters table.

### Insert into batch\_master table

When we will execute the following command: - node uploadUserBatch.js, an entry for the user batch upload will be made in the batch\_master table. The batch\_master table consists of the following columns: -

batch\_size

batch\_name

origin

If the batch file contains 10 users then the batch\_size will be entered as 10 in the database. The batch\_name can be any name that we want to give our batch upload. Origin indicates from where the request originated. In this case, as the request originated from node js, we have set the origin as “node”. The batch\_id that will be generated for the user batch upload, will also be referenced across the batch\_execution\_status and the batch\_exception tables.

### Insert into batch\_execution status table

When we execute the command: - node uploadUserBatch.js, an entry is inserted into the batch\_execution\_status table, for the user batch upload, with the is\_success column set as “Y”. The batch\_execution\_status table consists of the following columns: -

parameter\_name

parameter\_value

is\_success

batch\_id

The parameter\_name has been set as username which is being read from a constants file. Parameter\_value is the actual username (that has been inserted under the username column of the user\_masters table in the database). Is\_success column is set to “Y”, for each user, when that user record has been successfully inserted into the user\_masters table. In case of an exception, the is\_success column is set as “N”, for that particular user record, and an entry is inserted into the batch\_exception table for that user record. The batch\_id column references the batch\_id of the batch\_master table.

### Insert into batch\_exception table

In case an exception occurs, for a particular user record, during the user batch upload, then an entry will be inserted into the batch\_exception table for that user record. The batch\_exception table consists of the following columns: -

batch\_id

parameter\_name

parameter\_value

exception\_reason

exception\_desc

origin

The batch\_id column references the batch\_id of the batch\_master table. The parameter\_name has been set as username which is being read from a constants file. Parameter\_value is the actual username (that has been inserted under the username column of the user\_masters table in the database). Exception\_reason and exception\_desc columns have been set to null for now. Origin indicates from where the request originated. In this case, as the request originated from node js, we have set the origin as “node”. The records will be inserted into this table only when the exception method has been called, that is when an exception occurs during the insertion of records in the user\_masters table. If a particular record throws an exception, then an entry for that record will be made in the batch\_exception table, and then the user batch upload process will continue to insert the subsequent user records in the user\_masters table.

3.9 Authentication and Authorization

The application provides a ‘/login’ service. This service accepts username, password and tenant\_name. The service validates the user against the password entered. If the user is authenticated the services generates jwt token. The jwt token consists of username and permissions as the payload.

Request:

url -  <http://host:port/services/login>

method – post

postdata –

{

               "username":"admin",

               "password":"admin",

               "tenant\_name":"Tenant\_2"

}

Response:

{

    "msg": "login successful",

    "auth": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFkbWluIiwicGVybWlzc2lvbnMiOlsidmlldyIsImNyZWF0ZSIsIm1vZGlmeSIsImRlbGV0ZSJdLCJpYXQiOjE1MzM4MDU5NjcsImV4cCI6MTUzMzgwOTU2N30.6grY9CwM7yYSHJia2nAKO7qEQKRaPSaRSNpgbA7Yt6s"

}

The token obtained above is generated using jsonwebtoken package.

This package uses a certificate as secret. The path to certificate for each tenant is stored in tenant\_master table. In the application the certs are stored in /config/cert-store folder in pem format.

The other services expect the token to be part of authorization header. The service shall obtain the token from authorization header and shall decode it using jwt.verify method to obtain the claim. The claim shall have the username and its permissions (role). If logged\_in\_user has a permission for a particular action, the service shall invoke the mongo client to obtain the data. If the user does not have permissions than the service shall send 403 status.

3.10 Unit Testing and Coverage

A sample test for covering authorization js has been written using sinon.js and mocha as runtime.  Various dependencies were mocked.

/test/mock-authorization.js

*sinon.stub(jwt,"verify").yields(null,{username:'testUser',permissions:['create']});*

*sinon.stub(fs,"readFileSync").returns(new Object());*

*queryUtils['tenantData'] = [{tenant\_name:'test',cert:'/test/path'}]*

Run test using

npm test

get the coverage using

node\_modules/nyc/bin/nyc.js npm test

Sample out put-

test authorization

√ test if user was authorized

test authorization

√ test if user was not authorized, invalid role

test authorization

√ test if user was not authorized, invalid user

test authorization

√ test if err

4 passing (41ms)

=============================== Coverage summary ===============================

Statements   : 94.12% ( 32/34 )

Branches     : 83.33% ( 10/12 )

Functions    : 100% ( 3/3 )

Lines        : 94.12% ( 32/34 )

================================================================================