

Writing a good ETL flow with NodeJs



Daniele

Jul 27, 2019 · 4 min read

Last story was about if it's better to write your own ETL flow instead of using any of the tools available nowadays. in this story we will cover how to write a good quality ETL flow with javascript and Node js.

. . .



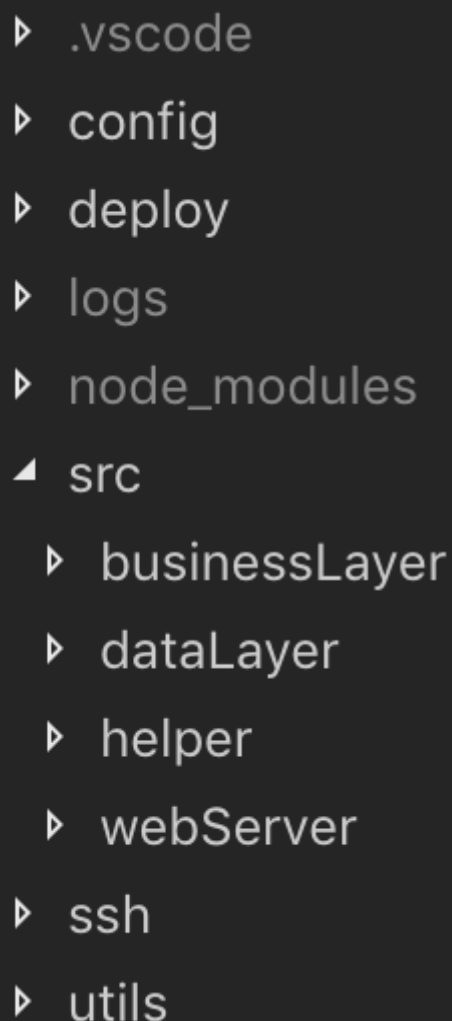
Photo by Shahadat Shemul on Unsplash

Requirements

- Node Js (use the version you're most familiar)
- Vs Code
- A Linux machine (Ubuntu or Debian should be fine)
- A database (We will use PostgreSQL 11 use whatever you like most)
- Pino an advanced logging script
- Sequelize this node module helps us for querying any database
- PM2 a NodeJs scheduler

Folder structure

A good folder structure is needed in order to give to each part of your software its importance and to have a mental organisation where to find the piece of code you need.



```
▶ .vscode
▶ config
▶ deploy
▶ logs
▶ node_modules
▲ src
  ▶ businessLayer
  ▶ dataLayer
  ▶ helper
  ▶ webServer
▶ ssh
▶ utils
```

This is the usual folder structure i use for my projects.

- **config** → Here goes all the configurations from database connections and node environments
- **deploy** → Here are all the deploy scripts, usually using ssh2 and git should do the work of deploying any changes
- **logs** → This folder will help you a lot in debugging
- **src** → Here is the heart of the ETL
- **src/businessLayer** → Here are all the business related behaviour of your software.
- **src/dataLayer** → This folder contains the mapping of your database column to Sequelize
- **src/helper** → All the scripts useful to obtain transformations should be here
- **src/webServer** → Needed for starting the ETL process
- **ssh** → We don't use password, we like keys
- **utils** → Here are all the utilities (Testing some code)



Photo by Pankaj Patel on Unsplash

Let's code!

Our starting point is **index.js**.

In this file you should put all the logics needed to avoid any incongruence for example:

- Check if all the required schemas are created
- Check if all the tables are created
- Check if all the stored procedures and functions are created

```
1  async function checkRequirements() {
2    await databaseName.query("create schema if not exists schemaName")
3    await databaseName.sync({ force: false }) //It will check if only the schema does not
4  }
5
6  async function checkTables() {
7    await databaseName.query("create table if not exists schemaName.tableName")
8    await databaseName.sync({ force: false })
9  }
```

index.js hosted with ❤ by GitHub

[view raw](#)

Once you've done those checks let's start with the first part of our loading.

In dataLayer you should put the database structure you have for example:

```
1  module.exports = function(sequelize, DataTypes) {
2    return sequelize.define('tableName', {
3      col1: {
4        type: DataTypes[dataType (Eg. INT, BIGINT, STRING, TEXT)],
5        allowNull: [true/false],
6        field: col1 //avoidable if equal to column name this allows the renaming of column
7      }
8    })
9  }
```

databaseModel.js hosted with ❤ by GitHub

[view raw](#)

For each table you have to create a raw import of your data without transforming it at first (maybe only a rename should be made at this point) with a simple select * query importing only the difference from the last import. A good check should be the lastInserted data and load only the data produced after that period.

Two are the ways to do this:

- Creating a SQL file containing the instructions, read the file and execute it
- Writing from code with Sequelize it should be faster but harder to explain to a non developer.

From Staging to DWH

This is the funnier part of the work, as long as you have created the database schema you should write the import code with some transformations (eg. dataTypes) and some simple computations (Eg. counting and summing)

Don't forget the keys

Every time you do an insert don't forget to drop the primaryKeys, every constraint and every index, this allows a faster insert and avoids errors.

Battling with existing data

Each time you start to import your data into dwh delete, before any insert, all the existing data matching the ids that are into staging, this allows to implicitly update the data instead to use huge updates that are slow.

So do something like this:

```
1 DELETE
2 FROM dwhTable
3 WHERE id IN (
4     SELECT ID
5     FROM stagingTable
6 );
```

fromSourceToStaging.js hosted with ❤ by GitHub

[view raw](#)

It is obvious that you need to delete every data in staging environment before importing from source in order to make this logic work.

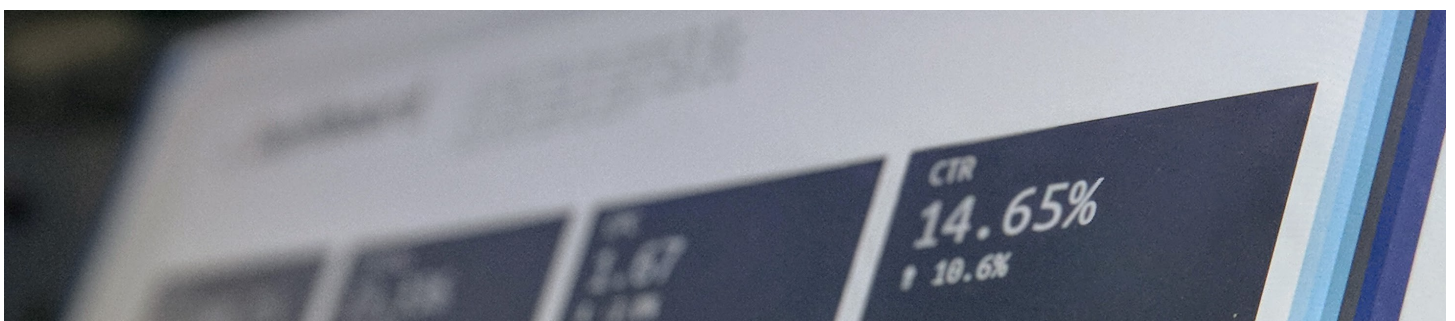




Photo by Stephen Dawson on Unsplash

The last part

The last but not the least here you create your data marts using SQL scripts and Sequelize to read the file and execute it.

PM2

With PM2 you can create a cron-like job that starts the nodeJs scripts at the time of your preference.

```
1  **
2  * Start etl process
3  * @param {object} schedule object represent node-schedule
4  */
5  module.exports = (schedule) => {
6
7      logger.debug('etl scheduled ' + FREQUENCY)
8
9      return schedule.scheduleJob(FREQUENCY, async () => {
10
11          try {
12              logger.debug('start etl through scheduler')
13
14              await startEtl()
15          } catch (error) {
16              exceptionLogger.fatal(error)
17          }
18      })
19  }
```

```
10
19     })
20 }
```

pm2.js hosted with ❤ by GitHub

[view raw](#)

With ubuntu just use:

```
pm2 start etl.js — name “my-etl”
```

and with PM2 list you can list all the jobs created in PM2.

Checking ETL status

Another good approach to check if the job is running well is to write each step into a loggingSql table structured like this for example:

```
1  -----
2  Id | Table name | start | end   | duration |
3  -----
4  1 | stagingTable | 00:00 | 00:10 | 00:10    |
```

table hosted with ❤ by GitHub

[view raw](#)

This is very useful for checking in real time what is doing the job:

```
1  async function loggerJobStartTime(jobName) {
2      return await loggerJobTable.models.etlJobLog.create(
3          {
4              jobname: jobName,
5              status: 'running',
6              starttime: Date.now(),
7              endtime: null,
8          },
9      )
10
11 }
```

joblog.js hosted with ❤ by GitHub

[view raw](#)

And at the start of each step update the table inserting the duration of previous step by doing a simple (endTime - startTime).

This is obvious an example, let me now in the comment what are your best practices and let's start a discussion.

[Nodejs](#) [Etl](#) [Data](#) [Database](#) [Data Engineering](#)

[About](#) [Help](#) [Legal](#)