Setup:

➢ **Introduction**:

- This setup is for backend and implemented in NodeJS.
- This setup provides various commands which helps developer to build folder structure, apis, middlewares, database configuration, etc… easily.
- This setup provides connection with MySQL database using sequelize.
- It provides error handler so that, whenever error generates, app doesn't get crashed. It will show the error, its reason and when error generated, along with that it will also show the line number and file path from where error generates.
- If defined port is busy then it asks to switch to the next nearer free port.
- It provides file upload solution.
- It provides the feature for api documentation.
- With 'npm start' command, server starts when it is called and using 'npm run dev' command, server starts automatically.
- It shows pending migration files if any and asks for migration.
- It can save database logs files and also show logs in console with date and time.
- It also shows what action is being performed in console like url, HTTP method, status, req-body, req-headers, etc…

➢ CLI commands

- framework init
  - ⇨ It installs the whole folder structure with required configuration files in your system. It asks for database configuration whether developer wants to do or not. If yes then it asks for development or production field and according to that, it asks for database connection information like username, password, database name, etc… and generates database.json file in config folder.

- framework create-module <moduleName>
  - ⇨ It creates the module in api folder which consists of controllers, middlewares and service folders and each folder has a sample file with dummy data and also routes.json file.

- framework create-api
  - ⇨ It creates the api. It asks for module name, in which developer wants to create an api and then it asks for path information like endpoint, HTTP method, controller file name and function name (separated by '.'), middleware file name and function name (separated by '.' and middlewares separated by ','), public or private endpoint, call from root or not, etc… It automatically creates routes.json file with given api's data and also creates controller file and function, middleware and global middleware files and functions in given module's folder which are specified by developer. If file already exists, then it appends the data.

- framework create-middleware
  - ⇨ It creates middlewares. It asks for module name, in which developer wants to create a middleware and api name, in which this middleware will be used. Then it asks for middleware file name and function name (separated by '.' and middlewares separated by ',') and it creates files and functions in middlewares folder of given module's folder, if file already exists, then it appends the data. It also adds the middlewares information provided by developer to routes.json.

- framework create-globalMiddleware
  ⇨ It creates global middlewares. It asks for module name, in which developer wants to create a global middleware and api name, in which this global middleware will be used. Then it asks for global middleware file name and function name (separated by '.' and middlewares separated by ',') and it creates files and functions in middlewares folder, if file already exists, then it appends the data. It also adds the global middlewares information provided by developer to routes.json.

- framework create-function
  ⇨ It asks for module level functions and global level functions. In module level function, it provides available modules and asks for creating new module to create function in that module. In global level function, it will create functions folder in root and will create the file. Then it will ask for function's file name and function name in filename.functionName format.

- framework create-service
  ⇨ It asks for module level services and global level services. In module level service, it provides available modules and asks for creating new module to create service in that module. In global level service, it will create services folder in root and will create the file. Then it will ask for service's file name and function name in filename.functionName format.

- framework db-config
  ⇨ It asks for database configuration whether developer wants to do or not. If yes then it asks for development or production field and according to that, it asks for database connection information like username, password, database name, etc… and generates database.json file in config folder.
- framework help
  ⇨ It shows the list of commands and its use.

➢ Folder Structure:

- api

  ⇨ api folder will have modules that will be created by developers, and in those modules, there will be controllers, middlewares and services folders and routes.json file.

  - moduleName

    -> In big projects, there may many routes and controllers and middlewares used. So to manage all the actions clearly, we should separate the things in modules. Ex. In E-Commerce website, there are many modules like shipping, payment, admin panel, user profile, products, etc… And each modules will have many routes like in shipping - remainingShipment, completedShipment, In products – addProduct, deleteProduct, editProduct, etc… So to manage all these routes and their actions and HTTP methods, we will separate these fields as a module.

    - controllers

      -> controllers folder will have controller files created by developers and the apis will be in object (key-value) format. For handling the error in api, it has setup.findErr(err) function, Developer just needs to pass error to that function and it will show the error information.

      Ex. module.exports = {
      <api>: () => {};
      }

    - middlewares

      -> middlewares folder will have middleware files used in apis that are declared in controller files, created by developers and the middleware is also in object (key-value) format.

      Ex. module.exports = {
      <middleware>: () => {};
      }

- services

  -> services folder will have service files used in apis that are declared in controller files, created by developers and the service is also in object (key-value) format. There is a global object named 'setup'.

  Ex. module.exports = {

  <service>: () => {};

  }

  To import service in api from service folder, we need to use setup.services['folderName']['fileName']['functionName']

- routes.json

  -> As name suggests, it defines routes. It has json data in which all the endpoints, HTTP method of each endpoint, what action it needs to take place when calling the particular endpoint, which middlewares particular endpoint will use, the endpoint is publicly accessed or not, endpoint should be called from root or not (here root will be the moduleName). These all informations will be saved in json object of json array.

  Ex: [{

  path: 'path/to/api',

  method: 'REQUEST_METHOD',

  action: '<ControllerName.ActionName>',

  middleware: ['<FileName.FunctionName>'],

  globalMiddleware: ['<FileName.FunctionName>'],

  public: true | false,

  root: true | false

  }]

- config
  - config.js

    -> This file has configurations for showing the data of which url, HTTP method, http version, status is being called with current date and time in colsole. It uses 'morgan' package.

    - config.json
      -> This file has json data in which developer can define there req-headers or req-body, where req-body defines what body is going to be passed when url is being called and req-headers defines the headers is used in calling the url, whatever developer wants to see in console when url executes, that developer needs to define there. Config.js will work on this data and it will show information in console.
    - database.json
      -> This file has json data of required information like username, password, database name, host, dialect etc... For connection with database in development and production area.
    - fileUpload.js
      -> This file has configuration for uploading any files to given location. This file uses 'multer' package. Developer just needs to write setup.store({LocationPath}) to store files and setup.uploadFile which is an object to define files.
- core
    - connection.js
      -> This file establishes the database connection using sequelize. It will ask developer for DB logs with date and time whether developer wants to save logs in files or show logs in console or both.
    - crons.js
      -> This file runs all the crons files of crons folder created by developer. If any cron is scheduled off and at particular time, it should get on and off, for that developer needs to write setup.crons['FileName']['cronName'] to call a particular cron.
    - moduleFunctions.js
      -> It has configuration for calling function files from function folder of given module. Developer just needs to write setup.moduleFunctions['folderName']['fileName']['functionName'](params1,params2,…,paramsN) to call the function. Functions folder can have multiple folders and files.
    - functions.js

-> It has configuration for calling function files from function folder of root folder. Developer just needs to write setup.functions['folderName']['fileName']['functionName'](params1,params2,…,paramsN) to call the function. Functions folder can have multiple folders and files.

- migrations.js
  -> This file handles all the migrations. If any migration is pending, then it shows the pending migration files to developer and asks to run.
- routes.js
  -> This file handles all the routes which are defined in routes.json and maps path to the api, middlewares, global middlewares which is declared in routes.json.
- moduleServices.js
  -> This file handles the configurations of services of given module. Developer can access the service files in api via setup.moduleServices['fileName']['serviceName']

- services.js
  -> This file handles the configurations of services of root folder. Developer can access the service files in api via setup.services['fileName']['serviceName']

- crons
  ⇨ This folder can have multiple cron files created by developer. Crons will be in object (key-value pair) format. Ex.:
  module.exports={
  <cron>: ()=>{}
  }

- db
  - migrations
    -> This folder includes multiple migration files.
  - models

-> This folder includes multiple model files.

- seeders

  -> This folder includes seeder files.

- dbLogs

  This folder includes database log files.

- functions

  ⇨ This folder holds multiple folders and multiple files for functions.

- middlewares

  ⇨ It holds middleware files which are common for many apis or modules. These middlewares are global middlewares.

- services

  ⇨ This folder holds multiple files for service.

- src

  - app.js

    -> This is the root file where the process starts. It imports all the required files and route calling starts from here. It includes error handler also which handles all the errors so that app doesn't get crashed. It has swagger configuration also for api documentation.

- uploads

  ⇨ This folder includes files uploaded by end user if developer has not set path for files storage.

- .env

  ⇨ This file includes port number defined by developer.

- .sequelizerc

  ⇨ This file defines the path of database folders and file like migrations, seeders, models and database.json.

➢ Where global object is used

- setup.functions['folderName']['fileName']['functionName'](params1, params2,…,paramsN)
  - ⇨ For importing the function from given path of functions folder of given module.
- setup.moduleFunctions['folderName']['fileName']['functionName'] (params1,params2,…,paramsN)
  - ⇨ For importing the function from given path of functions folder of root folder.
- setup.services['fileName']['functionName']
  - ⇨ For importing the services from services folder of given root folder.
- setup.moduleServices['fileName']['functionName']
  - ⇨ For importing the services from services folder of given module's folder
- setup.crons['fileName']['functionName']
  - ⇨ For importing vrons from crons folder.
- setup.findErr(<error>)
  - ⇨ It returns the reasons of errors in easy reading format.
- setup.store(<path>)
  - ⇨ It sets the path of uploading files.
- setup.uploadFile
  - ⇨ It defines the files which are going to upload.

➢ How to use this setup?

- As a beginner, first generate folder structure using ‘framework init’ command. It installs all the required packages and generates the folder structure.
- To create the modules, use ‘framework create-module’, it will generate the module folder in api which consists of cntrollers, middlewares, services folders along with files in it with dummy data, so that developer can get idea how to write apis or middlewares in those files and it also generates routes.json file.
- To create api, use ‘framework create-api’, it will ask all the information regarding to api such as endpoint, HTTP method, controller file and function, middleware files and functions, etc… and will create the files and functions also.
- To create middleware in particular api, use ‘framework create-middleware’, it will create middleware files and functions in given module and also adds the information in routes.json file.
- To create global middleware in particular api, use ‘framework create-globalMiddleware’, it will create global middleware files and functions and also adds the information in routes.json file.
- To create global services, use setup.services[‘fileName’][‘functionName’], to import service function from services folder of root folder to api.
- To create module level services, use setup.moduleServices[‘fileName’][‘functionName’], to import service function from services folder of module folder to api.
- To create global functions, use setup.functions[’folderName’][‘fileName’][‘functionName’](params), to import functions from functions folder of root folder to api.
- To create module level functions, use setup.moduleFunctions[’folderName’][‘fileName’][‘functionName’](params), to import functions from functions of module folder to api.
- To create crons use setup.crons[‘fileName’][‘functionName’], to import cron function from crons folder to api and you can start or stop that cron.

- To handle the api errors, in catch block, you can define ‘new Error()’ to a variable and can pass ‘setup.findErr(error)’ to show the reasons of errors. Ex. const err = new Error(err.name+’ ’+setup.findErr(error));
  err.status = 500; next(err);
- To upload a file, you can use ‘setup.uploadFile’, it will define the uploading files and with ‘setup.store(<path>)’, you can define a storage path of uploading files.
-
- If you want to see the req-body or req-headers which are being passed when calling the url, you can write that in config.json file of config folder.
- With ‘npm start’ command, server starts when it is called and using ‘npm run dev’ command, server starts automatically.

➢ TODOs

- We could make more commands for creating functions, services, crons.
- We can create a command for creating models, seeders and migration files instead of using sequelize command.
- We can create global object for importing the model files to controller files.
- We can put json web token configurations by default as it is very useful in most of the websites and also like that, the things which are mostly used in production level, that we should put by default, so that developer can directly use that, no need to do any configurations for that.