Documentation

Index

1. Brief of Concept

2. Workflow

3. Tech Stack

4. Folder Structure

5. Logic

# Brief of Concept

The NEST and CPT assessments use several pieces of data. Firstly, the question banks used for these two tests are stored in the Question collection in mongodb. Each question has multiple choices assigned a certain point. For NEST objective questions, only one correct option has 1 point while the rest have 0 points. For NEST and CPT subjective type questions, each choice contains points from 1 to 6. The formula’s for scaling the scores uniformly will be discussed later. NEST and CPT assessments need to be scored by different categories known as Modules. Each question is assigned to a particular Module and the final points for the respective modules provide the module score. After scoring the different assessments with their custom grading algorithm, a user score object is created in the UserScore collection in mongodb. All UserScore documents must contains the user\_id, the name, id and key of the assessment, a date that defaults to the current date, a plot type string and an array of module scores. The module scores array is the core of the user score. Each item in the array contains the score, name and id of the module. This model can be used for any assessment where the final score is in a tabular form. The module can be interpreted as the different sections or categories in the assessment and each module contains a total score. The module id can be specified by us and should be consistent for a given assessment. The plot type comes from the UserAssessment model and specifies the kind of graph to draw for the this score. Optionally, there can be a short description for each score. On starting any kind of assessment, a UserAssessment object is always created. This contains information about the assessment that the user has enrolled in and is used to check if the user is enrolled. If a userassessment object exists for a given user id and assessment id, then the user is redirected to the assessment page otherwise, they are denied access. The userassessment also contains useful stats about the assessment such as the number of attempts taken by the user, a completed field with a Boolean value, timestamps and assessment information. The user can retake an assessment multiple times, but the user assessment object is the same for every attempt. We simply increase the number of attempts and reset the completed field during a fresh attempt. For NEST and CPT only, there are some UserModule objects also being created. These are important to store information about the time spent so far on a particular module, the number of questions attempted, the total questions in the module and a status, showing pending or complete depending on if the user has answered all questions. This data is shown in the first page of the assessment. Each module contains a button to open the form containing questions related to that module. The previous answers are auto populated and the time spent is also updated with the user module data. For single page assessments, user module is not used. The user modules are created by fetching the modules in the assessment. We loop through all the modules and create a separate user module document for each, with the user’s id and assessment information added to it. For creativity assessments, the question bank used has a different format. I narrow the question bank types to the following- RatingStatement contains questions for assessments where there is one statement which needs to be rated according to a set scale. The LeftRightStatement collection contains questions where a user is given two statements- one on left, and one on right. The user has to pick a value closer to the side that they relate to the most. The responses of different assessments also vary in format. But, we can combine both the RatingStatement and LeftRightStatement response’s into a single UserResponse model. The style of these documents is that, they contain the user\_id, the name of the assessment, the id of the question, a category name and id optionally, and a numeric value. The same data can be interpreted differently in the specific assessment controllers. UserAnswer is the other response model type where we again store the user\_id, question\_id, module\_id, module\_name, a choice and a value. These are used for the multiple choice assessments. The choice is the option they choose such as ‘A’,’B’,’C’,etc. The value is the actual value of that choice. This field is not important for the scoring part. Next, we have the Answer model. This is the collection of answer banks for the assessments that required one. Namely, NEST relies on this to score the user on submission. The Divergent Thinking assessment is very different from the other creativity assessments, and so has it’s own set of models to store the responses and scores. Except this assessment, all user scores are put in the UserScore model. The creativity assessments which have only a few questions or less data are not using the database to fetch questions. Instead, I store the questions in the resources/json firectory on the server itself. We can fetch the entire JSON array containing the questions and send them to the client. Those assessments not using a database, also have some additional data on the server. The custom scoring algotihms will be discussed separately. There is one other model NESTFeedback, solely for NEST, although can be used for similar assessments. This collection is used for the report generation after the user has completed the test. The way we generate the report is that based on the range of score for a module, we fetch different feedback content and description. Thus, the model contains a min\_value, max\_value, module\_id, feedback and description fields. We query the collection to find the appropriate response for all the module scores and send them to the client. There are a few other models, unrelated to the assessment. The UserProfile is a model containing personal information of all users. It contains the location, qualifications, data of birth, img\_url for the profile picture, name, email, password, status, mobile, role and timestamps. This model also comes with static functions on the schema used for validation. The user’s without a profile picture are set a default avatar using an api url containing their initials in the query string. The password is always encrypted with bcrypt before saving to the database. While logging in, we compare the decrypted password with the encrypted password to authenticate the user. The other details in the user profile are straight forward. The Token model is used sparingly for situations like password reset. We generate a auto-time out token and email the user a url with this token in the query string. They are able to use this to reset the password. We may use this model for email change or password change in the future. The contact and chat and message models are not in use. Their purpose is for building a simple chat system in the website for communicating with other user’s or the customer support. The models/index.js exports all the models in this directory. It contains an object with key equal to the model name and value equal to the model. This allows us to easily import multiple models in a file from a single place. The config folder is important for setting up the services used by the app. S3.config.js initializes the AWS bucket used for storing profile pictures and exports methods to upload and retrieve the image from AWS server. Currently, the config files for multer, cookie session and redis are not being used. The strategy config files are used for passport js. They set up the functions for authentication. As for the frontend, I use server-side rendering using a templating engine known as EJS. The EJS files are all stored under the views directory. These contains the html for the frontend. There is little to no custom css as most of the styles come straight from Bootstrap 5. The routers directory contains the main business logic. The routers often import functions from the files in controller directory and have their own logic inside it as well. To organise the assessments, there are the following main sub routers- The enroll router is used for enrolling into any assessment. It creates the UserAssessment and UserModule, if there are modules, for the assessment. There are several middlewares before the routers, set up in the server.js file. The isAuth middleware restricts the users who are authenticated to access the content. We also save some local variables in the response object for later, such as loggedIn is set to true or false based on the state of authentication. The isEnrolled middleware checks if the user is enrolled before moving forward. It fetches the userAssessment object from the database and stores it into a user\_assessment local variable. This comes in handy in many assessments where we need to check the assessment keys and names. We do not need to make an additional db request. The assessment routers prefixes the assessment category name before the assessment keys. The assessment keys are the main form of identification of assessments on the server. They are custom set by me in the csv files. NEST and CPT are the keys for the corresponding assessments. For the creativity assessments, CP, CE etc are keys using the initials of the assessment. It is easier to set up the unique creativity routers based on the assessment key so we do not need to remember the ids of each one. The pattern that most of the creativity assessments use is the following- the ‘/questions’ get route, the ‘/submit’,’/save’ post routes. The questions route renders the questionnaire form for the assessment. The save route saves the current user answers in the db without submitting it, and the submit route saves as well as grades the user on submission of the assessment. They are then redirected to the scores page.

Data

* The static website data is contained in the csv files within the resources/csv directory. The files are migrated manually to the database by running the migration scripts provided in the migrations directory. The scripts should be run in the same order as the files appear within the migrations directory. The question banks are stored in the sub directory migrations/questionbanks. These must be run only once and they always stay the same. The categories and assessments may be updated and run again. The migrations do not affect the user data in the db, except the ClearUserTable.js migration.

Database

* The mongodb connection is iniitialised in the config/db.config.js module. It opens a database connection using the connection string specified in the environment variables. In production, the app uses a MongoDB Atlas cloud server to store its data. I have used the mongoose library to set up the db schemas and models, data validatation and queries.

Authentication

* Passport JS: Authentication is done with help of the passportjs library. It contains a local strategy and a google oauth strategy that allows the user to sign up and login with either their email and password, or through their Google account.

Multer

The multer package is used for

# Workflow

# Tech Stack

Node JS, Express JS

EJS

MongoDB

# Folder Structure

The app loosely follows the MVC pattern, that is, model view controller.

Folders:

1. Config/ - The top level config folder contains configuration and initialization files for libraries used in the project.
2. Controller/
3. Migrations/
4. Public/
5. Resources/
6. Services/
7. Views/
8. Server.js

# Logic