Postgres ORM

using postgres with node

Lesson Plan

- what is postgres
- Creating todo database with postgres
- Connecting to db from node
- what is ORM
- Creating our first model
- table relations
- What is a REST Server
- building our rest server
- authentication and authorization of REST Server
- Migrations

Lesson Summary:

https://www.nerdeez.com/articles/node/express-sequelize-postgres

Postgres

- open source database
- relational database
- SQL based
- Support Replication
- Indexes
- Table inheritance
- Triggers
- Stored procedures (user defined functions)
- Replication

Creating our first database

- install postgres
  - brew install postgresql
- First we need to create a db cluster
- creating a db cluster initialize a storage area for our db on disk
- db cluster contains many dbs
- db cluster is managed by a single instance of running database server
- you can create in a single location in your terminal
- create the cluster with the following command:
  - initdb -D /usr/local/pgsql/data
- to start the db server
  - pg_ctl -D /usr/local/pgsql/data start
- to create a db
  - createdb name-of-db
- to connect to the db
  - psql name-of-db

What is ORM

- object relational mapping
- ORM means transforming data to object oriented classes
- When abstracting ORM over database tables it means database manipulation like CRUD will be performed on classes and will not require sql queries
  - makes our code more readable
  - makes it easier to perform complex SQL queries
  - Makes our code database agnostic
- This means that if we have a table called users that holds our user account data, in ORM world we would probably have a User model where the table fields will be properties in the User model class.

Sequelize

- Sequelize is a promise based ORM for node
- It works with the following dbs
  - postgres
  - mysql
  - mssql
  - sqlite
- We create sequelize models that map to our tables.
- Sequelize will create the tables for us
- we perform queries on the model class, the queries will return instance of our model.
- Create update or delete is on the instances of our model

Install Sequelize

- We install sequelize using npm
  - npm install sequelize
- We will also need to install packages to connect to postgres and to connect sequelize to postgres
  - npm install pg pg-hstore

connect to database

- importing sequelize will bring us the sequelize class
- to create a new instance we need to pass the connection string to the database, this will bring us a sequelize connected to our database
- to verify that we are connected to the database you can call the **authenticate** method on the instance which will return a resolved promise if connection is success or a rejected promise on failed connection
- we will create it as a separate file and export the sequelize instance

Sequelize model

- lets create a users table
- our user will have only an email
- we create the models in a seperate file in the models directory
- We will need to use sequelize cache for the import this way we will have access to our **sequelize** instance
- so we are exporting a function which gets the sequelize instance and DataTypes
- we use sequelize.define to define the model where the first argument is the model name and the second is a specification of the fields
- After creating the model we can run **sync** to create the database
- to import we use **sequelize.import(path-to-user)**

Table associations

- you define associations between models (tables) by calling the following on the models
- 1:1 - belongsTo
- 1:m - hasMany
- m:m - belongsToMany

lets define a task model which is connected to the user one user can have many tasks

- a task has a title
- a task has a when

REST Server

- Representational state transfer
- rest is a communication protocol built on http
- It’s stateless nature makes each request isolated, and easily tested
- the response is usually json or xml (mostly json)
- the path describes what data we want
- the method describes what we want to do with that data

REST server for our tasks

- lets create a rest server to our tasks and users
- We will create CRUD on those models
- for the task we will populate the user field as well

```
Task.findAll({
            include: [
                {model: User}
            ]
        })
```

Validation

- We need to validate the data the user is sending
- We can define validation on the models
- lets validate on the user model that the email is valid email address and is required
- lets validate on the task model that title is alpha numeric up to 20 in length, and when is a valid date
-

Authentication & Authorization

- to make our application secure, we have to decide on every data resource, who can read it.
- so when the user requests data we need to ask two questions:
  - who is the user (authentication)
  - is the user allowed to access the resource he is requesting (authorization)
- How can we authenticate/authorize the user performing the request?

JWT

- stands for **Json Web Tokens**
- Json based open standard for creating access tokens
- The token is usually signed by private key in the server and only the server can decrypt it
- with JWT we can authenticate and authorize a user
- JWT contains claims about the token like privileges or date issued
- the server will create the token and send it to the user, the user will then pass the token on the headers
- The header will look like this:
    - **Authorization: Bearer <JWT token>**

JWT Structure

- JWT generally have 3 parts
  - header
  - payload
  - signature
- The header identifies which algorithm to use
  - **header = ‘{“alg”: “HS256”, “type”: “JWT”}’**
- The payload contains the claims to make, we can use this data to know the user that is making the request
  - **payload = ‘{“loggedInAs”: “Admin”, “iat”: 1422777777}’**
- The signature is calculated as the following:
  - **key = ‘secretkey’ // secret string of your choosing**
  - **unsignedtoken = encodeBase64Url(header) + ‘.’ + encodeBase64Url(payload);**
  - **signature = HS256(key, unsignedtoken);**
- So the token is the three parts seperated with ‘.’:
  - **encodeBase64Url(header) + ‘.’ + encodeBase64Url(payload) + ‘.’ + encodeBase64Url(signature)**

Migrations

- migrations describe changes in our database structure or data
- as our app evolves so will our database, it’s recommended to keep database structure declarative and define the changes by code.
- We define the change and the reverse change
- this will give us a recipe for creating our database and will allow us to quickly create a new db if its for testing or adding environments
- sequelize support migration for db change and seeders for data migrations
-

sequelize cli

- to run the migrations we will have to install sequelize-cli
  - npm install sequelize-cli
- create a migration file by running
  - sequelize migration:generate --name add-description
- populate the up with what you want to change in the database, you should return a promise. you can use the query interface to db changes
  - queryInterface.addColumn('tasks', 'description', Sequelize.TEXT)
- we can drop that column on the down method
  - return queryInterface.removeColumn({
    tableName: 'tasks'
    }, 'description');
- To run the migrations
  - sequelize db:migrate --url postgres://localhost:5432/todo

Summary

- in most server applications some sort of database is envolved and turning our db tables to orm makes rest server and query our db a breeze.
- Lets try to practice it ourselves

EX.

- create a sequelize user model containing email password and name
- all fields are string
- add validation for email on the email string
- add validation for max length of the name
- you should create a view to register and create a new user
- login is with email and password from the db
- create a REST server with user api to perform CRUD on the users table