ORM/Authentication
Sequelize/bcrypt

ORM

- What is ORM?
  ORM is a way to interact with a database, making it easy to create and update data. ORM stands for Object Relational Mapping. It is a way to map data to objects.
- Why Use ORM ?
  ORM allows dev to create and update data in a database using objects, making it maintain a clean and easy to use code.
- What is Sequelize?
  Sequelize is a library that allows you to interact with a database. It is a way to map data to objects.

ORM - Sequelize

- Sequelize is a Promise-based Node.js ORM for Postgres, MySQL, MariaDB, SQLite, and Microsoft SQL Server. It features solid transaction support, relations, eager and lazy loading, and read replication. You'll use the Sequelize package (Links to an external site.) to add Sequelize to your Node.js applications.

```
let mySampleObject = {
  name: 'John',
  age: 30,
  favoriteFood: 'pizza'
}
// create a table 'customers' of users and add mySampleObject to it
const Customers = sequelize.define('customers', {
  name: Sequelize.STRING,
  age: Sequelize.INTEGER,
  favoriteFood: Sequelize.STRING
})
Customers.sync()
  .then(() => Customers.create(mySampleObject))
  .then(() => console.log('Created customer'))
  .catch(err => console.log('Error creating customer', err))
```

Dotenv

- Dotenv : is a zero-dependency Node.js module that loads environment variables from a .env file into process.env, a Node.js property that returns an object containing the user environment. This allows developers to store user environment configuration—or the things that are likely to vary between different deployments (passwords, secrets, keys, etc.)—separately from their code.

```
const dotenv = require('dotenv')
dotenv.config()
```

bcrypt

- bcrypt: is a Node.js library that allows you to hash passwords. Hashing is the process of taking input and using a mathematical formula to chop and mix it up to produce an output of a specific length. Hashing is a one-way function, meaning that it can easily convert input to a fixed-size output, but it is difficult to invert, or convert in the opposite direction. This attribute allows developers to secure passwords when authenticating users for their applications.

FOLDER STRUCTURE

- The following is a list of the folders and files that you will need to create to get started with the project.
- The root folder is the folder that contains the `package.json` file.
- The `src` folder is the folder that contains the JavaScript files.
- The `model` folder is the folder that contains the SQL files.
- The `public` folder is the folder that contains the static files.
- The `.env` file is the file that contains the environment variables.
- The `routes` folder is the folder that contains the routes.
- The `config` folder is the folder that contains the configuration files.
- The `db` folder is the folder that contains the database files.

Sequelize model

- in Sequelize's model class we essentially our own JavaScript class and define the columns, data types, and any other rules we need the data to adhere to.

Create Sequelize connection

- in the `config` folder, create a file called `connection.js` and add the following code:

```
const Sequelize = require('sequelize')
const connection = new Sequelize('database', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql',
})
```

Add a model user to the connection

- in the `model` folder, create a file called `user.js` and add the following code:

```
const Sequelize = require('sequelize')
const connection = require('./connection')
const User = connection.define('user', {
  name: Sequelize.STRING,
  email: Sequelize.STRING,
  password: Sequelize.STRING,
})
```

Model Configurations

- Model is initialized with the connection.define method. The first argument is the name of the table. The second argument is an object that defines the columns and data types. The third argument is an object that defines the model's configuration.

colum definitions

- The columns are defined by an object with the column name as the key and an object with the data type as the value.
- value object properties
  - `type: DataType.Integer` is the data type for an integer.
  - `allowNull: false` is the default value for a column.
  - `primaryKey: true` is the default value for a column.
  - `autoIncrement: true` is the default value for a column.
  - `unique: true` is the default value for a column.
  - `defaultValue: null` is the default value for a column.
  - `validate: {isEmail: true}` is the default value for a column.

models config keys

- `timeStamps`: true or false. If true, sequelize will automatically add the current date and time to the values being saved. If false, sequelize will not add a value. Default is true.
- `freezeTableName`: true or false. If true, sequelize will not try to alter the table name to the model name. Default is false.
- `underscored`: true or false. If true, sequelize will add an underscore character before each of the model's attributes. Default is false.
- `tableName`: string. The name of the table. Default is the model name.
- `paranoid`: true or false. If true, sequelize will add a deletedAt timestamp for soft-deleted models. Default is false.
- `createdAt`: string. The name of the createdAt attribute. Default is `created_at`.

sequelize `findAll` query methods

Sequelize provides a number of methods that allow you to query your database. The following are the available methods:

- `findAll`: returns all rows from a table. which has has attributes like
  - `exclude: ['password']`: excludes the password column from the result.
  - `include: [{model: User, attributes: ['name']}]`: includes the name column from the User table in the result.
  - `limit: 10`: limits the result to 10 rows.

```
User.findAll({
  exclude: ['password'],
  include: [{model: User, attributes: ['name']}],
  limit: 10
})
.then(users => console.log(users)) // returns an array of user objects
.catch(err => console.log(err))
```

sequelize `findOne` query methods

- `findOne`: returns a single row from a table. which has has attributes like
  - `exclude: ['password']`: excludes the password column from the result.
  - `include: [{model: User, attributes: ['name']}]`: includes the name column from the User table in the result.
  - `limit: 10`: limits the result to 10 rows.

```
User.findOne({
  exclude: ['password'],
  include: [{model: User, attributes: ['name']}],
  limit: 10
}) // returns a user object\
.then(user => console.log(user))
.catch(err => console.log(err))
```

sequelize `create` query methods

- `create`: creates a new row in a table. which has attributes like

```
User.create({
  name: 'John Doe',
  email: 'JohnDoe@example.com'
})
.then(user => console.log(user))
.catch(err => console.log(err))
```

sequelize `update` query methods

- `update`: updates a row in a table. which has attributes like
- additionally, you can also pass an object with the attributes to update.
- to update user with id 45 we `where id = 45 we add another argument with the id.

```
User.update({
  name: 'John Doe',
  email: 'j@.com',
  password: '123456'
}, {
  where: {
    id: 45
  }
})
.then(user => console.log(user))
.catch(err => console.log(err))
```

sequelize `destroy` query methods

- `destroy`: deletes a row in a table. which has attributes like we an object with the attributes to delete.

```
User.destroy({
  where: {
    id: 45
  }
})
.then(user => console.log(user))
.catch(err => console.log(err))
```

other sequelize methods

- `count`: returns the number of rows in a table. `TbaleName.count( {where: {name: 'qasim'}} )`
- `max`: returns the maximum value of a column. `TableName.max('age', {where: {name: 'qasim'}} )`
- `min`: returns the minimum value of a column. `TableName.min('age', {where: {name: 'qasim'}} )`
- `sum`: returns the sum of values in a column. `TableName.sum('age', {where: {name: 'qasim'}} )`
- `avg`: returns the average value of a column. `TableName.avg('age', {where: {name: 'qasim'}} )`
- `findOrCreate`: returns an array with the first element being the instance of the model if it was found, or a new instance if it wasn't. `TableName.findOrCreate({name: 'qasim'})`

Authentication - bcrypt

`bcrypt` : is a library that allows you to hash passwords.
it takes in a password, hashes it using the bcrypt algorithm which is a one way encryption algorithm and returns a hash.
The algorithm is designed to be resistant to dictionary attacks, and works like this

`password` come in -> `bcrypt.hash(password, salt)` -> `hash` is the hashed password and `salt` is a random string that is used to hash the password.

Sequelize hooks and bycrypt

Hooks (also known as lifecycle events), are functions which are called before and after calls in sequelize are executed. For example, if you want to always set a value on a model before saving it, you can add a `beforeUpdate` hook. some common hooks

- `beforeCreate`: called before a new instance is created.
- `afterCreate`: called after a new instance is created.
- `beforeUpdate`: called before an existing instance is updated.
- `afterUpdate`: called after an existing instance is updated.
- `beforeDestroy`: called before an existing instance is destroyed.

```
User.define({
  email: {
    type: DataType.STRING,
    allowNull: false,
    validate: {
      isEmail: true
    }
  },
  password: {
    type: DataType.STRING,
    allowNull: false,
    validate: {
      length: {
        min: 6
      }
    }
  }
}, {
  hooks: {
    beforeCreate: (user, options) => {
      user.password = bcrypt.hashSync(user.password, 10)
    }
  }
})
```

Association and References in Sequelize

- we can associate a model with another model
- let say we want to associate comments with a user. we can do this by using the `belongsTo` method.

```
Comment.belongsTo(User)
```

- we can also use the `references` method to create a foreign key in the table.

```
Comment.references(User, 'userId')
```

- these relationships are defined in the `associations` object. which is an object that contains the association methods.
- some of the most common associations are:
  - `hasOne`: a one-to-one relationship.
  - `hasMany`: a one-to-many relationship.
  - `belongsTo`: a many-to-one relationship.
  - `belongsToMany`: a many-to-many relationship.

Association

- we can use the `associations` object to define associations. Let's say we have two models, A and B. Telling Sequelize that you want an association between the two needs just a function call:

```
const A = sequelize.define('A', /* ... */);
const B = sequelize.define('B', /* ... */);

A.hasOne(B); // A HasOne B
A.belongsTo(B); // A BelongsTo B
A.hasMany(B); // A HasMany B
A.belongsToMany(B, { through: 'C' }); // A BelongsToMany B through the junction table C
```

Associations - hasOne

- Take users of youtube, each user has one profile. We can use the `hasOne` method to define a one-to-one association.

```
User.define({
  email: {
    type: DataType.STRING,
    allowNull: false,
    validate: {
      isEmail: true
    }
  },
  password: {
    type: DataType.STRING,
    allowNull: false,
    validate: {
      length: {
        min: 6
      }
    }
  }
})
Profile.define({
  name: {
    type: DataType.STRING,
    allowNull: false,
    validate: {
      length: {
        min: 6
      }
    }
  }
})
User.hasOne(Profile)
```

Associations - hasMany

- Take users of youtube, each user has many comments. We can use the `hasMany` method to define a one-to-many association.

```
Comment.define({
  text: {
    type: DataType.STRING,
    allowNull: false,
    validate: {
      length: {
        min: 60
      }
    }
  },
  user_id{
    type: DataType.INTEGER,
    allowNull: false
    references: {
      model: User,
      key: 'id'
    }
  }
})
User.hasMany(Comment) // User HasMany Comment
Comment.belongsTo(User) // Many comments belong to one user
```