Object-Relational Mapping (ORM)

- A way to access and manipulate your database without having to write SQL queries
- ORMs generate SQL queries for you based on your JavaScript data model
  - Allows you to focus mainly on JavaScript code
  - This can cut down on SQL typos and mistakes and speed development

Sequelize

- Promise based ORM
- Read from, write to, and modify PostgreSQL tables with JavaScript

Adding Sequelize to your app

```
npm install --save sequelize
```

Connecting:

- Connection string is similar to pg
- Uses a constructor to create an instance of Sequelize

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('postgres://user:password@localhost/my_db');
```

Defining Models in Sequelize

- Instead of defining tables, we define models using Javascript objects
- Each property is defined with a datatype defined by Sequelize

```
//let's take another look at `hats` from the Postgres lecture
//define a `hat` model with the following properties:
var Hat = sequelize.define('hat', {
    //create name and material as strings,
    name: Sequelize.STRING,
    material: Sequelize.STRING,
    //height as an integer,
    height: Sequelize.INTEGER,
    //and brim as a true/false
    brim: Sequelize.BOOLEAN
});
```

Keeping models in sync

- Once a model is defined, use `.sync()` to ensure the table exists
- This is like running `CREATE TABLE` if the table does not exist
- Remember, Sequelize uses promises: once the model exists, you can chain `.then()` calls

```
//using the hat definition from the previous example,
Hat
    //ensure the table exists,
    .sync()
    .then(function(){
        //`Hat` is now ready to be used.
    })
```

Creating models

- `Model.create()` takes in an object that contains key/value pairs that map to the model definition
- Then, Sequelize builds and executes a SQL query based on the parameters supplied

```
Hat.create({
    name: 'cowboy',
    material: 'straw',
    height: 4,
    brim: true
});
```

- becomes:

```
INSERT INTO "hat" ("id", "name", "material", "height", "brim")
    VALUES (DEFAULT,"cowboy", "straw", 4, 1) RETURNING *;
```

EXERCISE: Create Your Models With Sequelize

Create a new database or, alternatively, DROP the tables you've created so far and re-create them with JavaScript and sequelize as we did with the Hat example.

```
DROP TABLE users;
DROP TABLE posts;
```

Finding all instances

- Using `Model.findAll()`, you can retrieve all instances of a given model

```
Hat.findAll().then(function(rows) {
    for(var i = 0; i < rows.length; i++) {
        var columnData = rows[i].dataValues;
        var name = columnData.name;
        var brim = columnData.brim;
    }
};
```

- becomes:

```
SELECT "id", "name", "material", "height", "brim" FROM "hats" AS "hat";
```

Finding models by ID

- Using `Model.findById(id)`, you can retrieve a specific instance of a given model by its primary key

```
Hat.findById(id).then(function (row) {
    var name = row.dataValues.name;
    var brim = row.dataValues.brim;
});
```

- becomes:

```
SELECT "id", "name", "material", "height", "brim" FROM "hats" AS "hat" where "hat"."id" = 1;
```

EXERCISE:

Add Express routes to find and return all posts, users

You are creating 2 routes, one for posts, one for users

**Review: **

```
app.get('/posts', function (req, res) {

  // your code here

});
```

Queries with Conditions

- Using `Model.findAll()` or `Model.findOne()`, you can supply conditions to your query to limit the records they return.

```
Hat.findAll({
    where: {
        brim: true
    }
})
.then(function(rows){
    for(var i = 0; i < rows.length; i++) {
        var columnData = rows[i].dataValues;
        var name = columnData.name;
        var brim = columnData.brim;
    }
});
```

- becomes:

```
SELECT "id", "name", "material", "height", "brim" FROM "hats" as "hat" where "hat"."brim" = 1;
```

LIKE and ILIKE Queries

- `LIKE` is a special SQL clause that does a simple match on substrings
- Uses a % sign as a wild card
- `ILIKE` is a a case-insensitive `LIKE`

Reference: http://docs.sequelizejs.com/en/v3/docs/querying/#operators

In SQL:

```
SELECT * FROM Hats WHERE name LIKE '%tetson' -- will match text ending in "tetson" like "Stetson"

SELECT * FROM Hats WHERE name ILIKE 'cow%' -- will match "Cowgirl" "Cowboy" "cowlick" etc

SELECT * FROM Hats WHERE name ILIKE '%cow%' -- matches anywhere in field, "Scowl" "Bellcow" etc
```

LIKE and ILIKE in Sequelize

```
Hat.findAll({
    where: {
        name: {
            iLike: 'cow%';
        }
    }
})
.then(function(rows){
  // work with returned data
}
```

EXERCISE:

Add an Express route to find and return all posts containing a search term

Hint:

```
/*
Use a query string parameter for the search term like:
   /posts/search?term=sometext
*/
app.get('/posts/search', function (req, res) {

  var query  = req.query.term;

  // your query code here

});
```

Updating your Model's Data

- Using a combination of `Model.findOne()` and `modelInstance.update()`, you can update your model data by passing an object of key/value pairs

```
Hat.findOne({
    where: {
        name: 'cowboy'
    }
})
.then(function(hat){
    hat.update({
        height: 3
    });
})
```

- becomes:

```
UPDATE "hats" SET "height"=3 WHERE "id" = 1
```

Exercise

- Compare/contrast your `node-postgres` assignment to Sequelize.
- What are some advantages / disadvantages of each?
- If we were to take Sequelize as a more evolved manifestation of the `node-postgres` assignment, what has changed?

Table association: One to Many

- Allows for making and maintaining foreign keys using Sequelize
- Defines relationships between entities
- The one-side of the relationship has a `.createModel()` method that automatically maintains the relationship for you

```
var Sequelize = require('sequelize');
var sequelize = new Sequelize('postgres://user:password@localhost/my_db');

//use the same hat definition as before
var Hat = sequelize.define('hat', {
    name: Sequelize.STRING,
    material: Sequelize.STRING,
    height: Sequelize.INTEGER,
    brim: Sequelize.BOOLEAN
});

//define a simple person model
var Person = sequelize.define('person', {
    name: Sequelize.STRING
});

//a person can have many hats...
Person.hasMany(Hat);
//... but a hat belongs to a single person.
Hat.belongsTo(Person);

sequelize
    //sync the models
    .sync()
    .then(function(){
        //then create a person
        //turns into INSERT INTO "people" ("id", "name") VALUES (DEFAULT, 'Jane Smith')
        return Person.create({
            name: 'Jane Smith'
        })
    })
    .then(function(person){
        //then create a hat for that person
        //turns into INSERT INTO "hats" ("id", "name", "material", "height", "brim", "personId")
        //  VALUES (DEFAULT, 'cowboy', 'straw', 3, true, 1) RETURNING *;
        return person.createHat({
            name: 'cowboy',
            material: 'straw',
            height: 3,
            brim:true
        });
    });
```