# Sequelize:

## A Postgres ORM

# How We Use Postgres Now

- The pg module is used to create a query pool to the database
- Tables are manually created outside of our code, and we rely on them being there
- We access the database by writing SQL queries as strings, sometimes inserting variables in to those queries
- All of our code relies on the database being Postgres

# Using an ORM Instead

- An ORM is an **Object Relational Mapping**, a way of using classes and objects as abstractions from the way the underlying data is accessed
- Databases are a great use case for ORMs, because you want the concept of what you're doing to work in any context, not just in Postgres
  - i.e. "Get John from the database" rather than "`SELECT * FROM people WHERE name='john'`"
- Our BulletinBoard or Grocery singletons were basic examples of us writing an ORM, but it would be nice if there was a more sophisticated way of having them be made for us...

# Introducing Sequelize

- Sequelize is an ORM that sits on top of pg
- It makes accessing our database look less like SQL, and more like traditional object oriented code
- We'll be able to define models that represent our tables, and have sophisticated objects with methods for our rows, not just basic data
- It also means we could have our same code work for any database, not just Postgres

# Sequelize: Getting Started

```
# Install the dependency
npm install --save sequelize
```

- Sequelize is an npm package like any other that you'd install

```
const Sequelize = require("sequelize");
const sql = new Sequelize(/* ...connection info... */);
```

- The module it provides is a class that we make instances of

# Sequelize: Initialization

- You can create a new Sequelize instance in one of two ways:
  1. A Database URL, like the one Heroku gives us
  2. A set of arguments, just like the pg.Pool class
- Like pg.Pool, we should be making a module for this so we only have to make one instance in our whole app
- Normally these would be `process.env.KEY` arguments, but examples would be:

```
// Database URL
const sql = new Sequelize("postgres://user:pass@localhost:5432/dbname");

// Object Arguments
const sql = new Sequelize({
    database: "dbname",
    username: "username",
    password: "password",
    host: "localhost",
    port: 5432,
    dialect: "postgres", /* DON'T FORGET ME */
});
```

# Sequelize: Defining Models

- We don't directly run queries with Sequelize, we define models for each table
- For each column in the table, we define what type it is with `Sequelize.TYPEs`, and add constraints as keys to an object

```
// models/users.js
const Sequelize = require("sequelize");
const sql = require("../util/sql");

module.exports = sql.define("user", {
    /* SERIAL PRIMARY KEY */
    id: {
        type: Sequelize.INTEGER,
        autoIncrement: true,
        primaryKey: true,
    },
    /* VARCHAR(16) NOT NULL */
    username: {
        type: Sequelize.STRING(16),
        allowNull: false,
    },
    /* TEXT, No constraints */
    bio: Sequelize.TEXT,
});
```

# Sequelize: Initializing Models

- Before using the models, you'll want to make sure it's `sync()`ed. Doing so in app.js is a good place.
- This will create the table if it doesn't exist, or do nothing if it does. This means no more manually running `CREATE TABLE`!

```
require("./models/users");
require("./models/posts");
const sql = require("./util/sql");
const app = require("express")();

sql.sync().then(function() {
    console.log("Database initialized!");

    app.listen(process.env.PORT || 3000, function() {
        console.log("App is listening!");
    });
});
```

*If you ever change a model, you'll need to either manually alter / delete the table, or use* **`sync({ force: true })`**

# Sequelize: Model Table Behavior

- The `define` function's first argument isn't a 1:1 mapping with the name of the table
  - Sequelize will create a table with a plural name by default, so "user" becomes "users"
- Sequelize will also add updatedAt and createdAt rows to your table, and set them accordingly when you make new rows
- These defaults are great going forward, but may mess with any projects you've made before
- Fortunately, these can be overridden with a third `options` argument:

```
sql.define("table", { /* columns */ }, {
    // Don't add createdAt, updatedAt columns
    timestamps: false,
    // Hard code the table name instead of dynamically making it plural
    tableName: "my_cool_table",
    // OR (instead of tableName) just use the model's name as the table name
    freezeTableName: true,
});
```

# Sequelize: Using Models (SELECT)

- Rather than write SELECT queries, we'll use the model's `find*()` functions
- The promise resolve returns either one row, or an array of them

```
const Users = require("./models/users");

// Get all users
Users.findAll().then(function(users) {
    console.log("There are " + users.length + " users");
});

// Get a user by id
Users.findById(123).then(function(user) {
    console.log("User 123's username is " + user.get("username"));
});

// Get all users who are 99
Users.findAll({ where: { age: 99 } })
    .then(function(users) { /* ... */ });

// Get one user (or the first of many) who matches a condition
Users.findOne({ where: { username: "searchForMe2017" } })
    .then(function(user) { /* ... */ });
```

# Sequelize: Using Models (INSERT INTO)

- Rather than write `INSERT INTO` queries, we'll use the `create*()` methods to make new rows
- The promise resolve doesn't return anything about the creation

```
const Users = require("./models/users");

// Create one new user
Users.create({ username: "newDude1987" })
    .then(function() { /* ... */ });

// Bulk create
Users.bulkCreate([
    { username: "newPerson1" },
    { username: "newPerson2" },
]).then(function() { /* ... */ });

// Search for something, and if it doesn't exist, create it
Users.findOrCreate({ username: "donJuan" })
    .then(function(user, wasCreated) { /* ... */ });
```

# Sequelize: Using Models (UPDATE)

- Rather than write `UPDATE` queries, we'll use the `update()` method to make changes to existing rows
- We get back an array that has `[numRowsUpdated, rows]` in the promise resolve

```
const Posts = require("./models/posts");

// Update all posts to be private
Posts.update({ public: 0})
    .then(function(res) { /* ... */ });

// Update one post to make it public
Posts.update(
    { public: 1 },
    { where: { id: 827 } },
).then(function(res) { /* ... */ });
```

# Sequelize: Using Models (DELETE)

- Rather than write `DELETE` queries, we'll use the `destroy()` method to remove existing rows
- We get back the number that were destroyed in the promise resolve

```
const ShoppingCart = require("./models/shoppingCart");

// Empty the shopping cart for one user
ShoppingCart.destroy({ where: { userid: 3819 }})
    .then(function(numDestroyed) { /* ... */ });

// Remove one item from a user's shopping cart
ShoppingCart.destroy({
    where: {
        userid: 3819,
        itemid: 448173,
    },
}).then(function(numDestroyed) { /* ... */ });
```

# Sequelize: Using Rows

- Many of the model's promises resolve with a row or an array of rows
- Rows are not just simple objects that have the column value as a key on it
- They have a `.get("column")` function that can turn the raw column value in to a more useful value
- For instance, a timestamp column could come back as a Javascript `Date` object instead of a string
- Likewise, rows have a `.set(value)` function for updating that row

# Sequelize: Using Rows (code)

```
// models/purchases.js
module.exports = sql.define("purchase", {
    id: { /* ... */ },
    price: {
        type: Sequelize.INTEGER,
        get: function() {
            // Turn the integer price in to a string (i.e. 995 becomes "$9.95")
            return "$" + (this.getDataValue("price") / 100);
        },
    }
});

// somewhere else in code
Purchases.findById(123).then(function(purchase) {
    console.log("Purchase cost " + purchase.get("price"));
});
```

# Sequelize: Conditional WHERE

- The previous examples only showed the `where` arguments as being exact matches
- However, SQL (and Sequelize by proxy) can do much more when it comes to filtering
- Where column arguments can also be comparisons for things like greater than, less than, contains etc.
- This is done by passing an object instead of a value, where they key indicates which comparison should be done

*Conditional WHEREs, From the Sequelize findAll docs*

```
Model.findAll({
  where: {
    // You would normally only use one or two of these
    id: {
      $and: {a: 5}           // AND (a = 5)
      $or: [{a: 5}, {a: 6}]  // (a = 5 OR a = 6)
      $gt: 6,                // id > 6
      $gte: 6,               // id >= 6
      $lt: 10,               // id < 10
      $lte: 10,              // id <= 10
      $ne: 20,               // id != 20
      $between: [6, 10],     // BETWEEN 6 AND 10
      $notBetween: [11, 15], // NOT BETWEEN 11 AND 15
      $in: [1, 2],           // IN [1, 2]
      $notIn: [1, 2],        // NOT IN [1, 2]
      $like: '%hat',         // LIKE '%hat'
      $notLike: '%hat'       // NOT LIKE '%hat'
      $iLike: '%hat'         // ILIKE '%hat' (case insensitive)  (PG only)
      $notILike: '%hat'      // NOT ILIKE '%hat'  (PG only)
      $overlap: [1, 2]       // && [1, 2] (PG array overlap operator)
      $contains: [1, 2]      // @> [1, 2] (PG array contains operator)
      $contained: [1, 2]     // <@ [1, 2] (PG array contained by operator)
      $any: [2,3]            // ANY ARRAY[2, 3]::INTEGER (PG only)
    },
    status: {
      $not: false,           // status NOT FALSE
    }
  }
})
```

# Additional Reading

- Sequelize Documentation
  - Getting Started
  - Defining Models
  - Using Models (Incomplete, only covers find* fns)
  - All Model Functions (Has what Using Models doesn't)
- Sequelize Express Example Project