# Sequelize Intro

## Summary

In this lab, you'll continue using the database we set up during the demo for Clark County Construction. You'll be writing in a new app though, this time one that's for admin.

## Setup

. Download the exercise materials from Frodo.

. Navigate to the folder in your terminal.

. Run npm install to get the project's dependencies installed

. Open your code and begin!

## Instructions

### Step 1: Dotenv

For our server files to work, we're going to need ***dotenv***.

. Install 'dotenv' using npm install dotenv

. Create a file at the root of your project called .env

. In that file, create a SERVER_PORT variable and set it to **8765** (it needs to be this number or you'll have to change all the front end code)

. Additionally, create a CONNECTION_STRING variable and set it to your ***URI*** from bit.io (same as the one you used in the demo)

. At the top of controller.js, require in the dotenv package and call the config method (you can ignore the other code in there until later steps)

. Optional: destructure CONNECTION_STRING from process.env in controller.js

### Step 2: Sequelize

In order to make queries to our database, we'll need Sequelize.

. Install Sequelize and its dependencies using npm install sequelize pg pg-hstore

. In controller.js, require the sequelize package and save it to a variable called ***Sequelize***

. Initialize a new Sequelize instance, passing in your connection string and an object that looks like this:

```
{
  dialect: 'postgres',
  dialectOptions: {
     ssl: {
        rejectUnauthorized: false
     }
  }
}
```

## Step 3: Getting Client Information

During the demo, we seeded our databases with lots of information. Now that we're working on the admin side of things, we want a way to see all of our clients and their contact info. The front end is set up to receive an array of client info that contains data from both the cc_users and cc_clients tables, so we'll need to use join.

. In controller.js's export object, write a new function called getAllClients (make sure it accepts req & res)

. Using sequelize.query query your database for all the columns in both cc_users and cc_clients joining them where the user_id column matches

. Handle the promise with .then() passing in a callback: dbRes => res.status(200).send(dbRes[0]) (you can also add a .catch)

. In index.js, comment line 20 back in (this line: app.get('/clients', getAllClients))

. Run nodemon (make sure you're in the right directory, also you can keep it running for the duration of the lab)

. Open clients.html (public folder) in your browser

. You should be seeing your client information!

## Step 4: Pending Appointments

If you head to the home page of CCL Admin Portal, you'll see sections for Pending, Upcoming, and Past Appointments. Pending appointments are those that have been requested but not yet approved. Let's set up a query for those now.

. In controller.js, write a new function called getPendingAppointments

. Using sequelize.query query your database for all appointments that are not approved (approved = false) and order them by date with the most recent dates at the top.

. Handle the promise with .then() passing in a callback: dbRes => res.status(200).send(dbRes[0]) (you can also add a .catch)

. In index.js, comment line 23 and 24 back in (these lines: app.get('/pending', getPendingAppointments), and app.get('/upcoming', getUpcomingAppointments))

. In home.js (public folder), comment line 107 back in (this line: getPendingAppointments())

. Navigate to home.html in your browser

. You should be seeing pending appointments!

## Step 5: Past Appointments

Next, you'll be writing a query that's similar to the triple join query from getUpcomingAppointments, reference it as you write yours. This one will get all of the past appointments.

. In controller.js, write a new function called getPastAppointments

. Using sequelize.query query your database for the following columns from their respective tables **cc_appointments**: appt_id, date, service_type, notes. **cc_users**: first_name, last_name. Reference the getUpcomingAppointments function to see how to join all the information together (you'll need all the same tables again). Make sure to select only rows where both the approved and completed values are true. And order the results by date with the most recent at the top.

. Handle the promise with .then() passing in a callback: dbRes => res.status(200).send(dbRes[0]) (you can also add a .catch)

. In index.js, comment line 25 back in (this line: app.get('/appt', getPastAppointments))

. In home.js (public folder), comment lines 108 and 109 back in (these lines: getUpcomingAppointments() and getPastAppointments())

. Navigate to home.html in your browser

. Now you should be able to see all the past appointments as well as the pending ones.

## Step 6: Approving Appointments

Part of this function has been given to you because it's taking care of auto-assigning employees to appointments for us. Finish it up with one query and the 'Mark Approved' buttons on the front end should work.

. In the approveAppointment function, delete *****YOUR CODE HERE**** and replace it with a query that updates the appointments table. It should set approved to equal true where the appt_id matches the one coming from the req.body, which has been destructured for you as apptId.

. In index.js, comment line 26 back in (this line: app.put('/approve', approveAppointment))

. Navigate to home.html in your browser

. Test out the 'Mark Approved' button!

**No appointments to approve?** There should be some appointments waiting for you from the demo. If there are not, start up your demo app again and request a few. Since they run on different ports, you can actually have your demo app and your lab app running at the same time.

## Step 7: Completing Appointments

Last but not least, let's make the 'Mark Completed' button work.

. In controller.js, write a new function called completeAppointment

. Using sequelize.query query your database to update the appointments table. It should set completed to equal true where the appt_id matches the one coming from the req.body. (You can destructure apptId like in the previous step or use req.body.apptId.)

. Handle the promise with .then() passing in a callback: dbRes => res.status(200).send(dbRes[0]) (you can also add a .catch)

. In index.js, comment line 27 back in (this line: app.put('/complete', completeAppointment))

. Navigate to home.html in your browser

. Your whole page should work now! Once pending appointments are approved, they should move to the upcoming section, and once those are complete they should move to the past section.

### Push your code to GitHub!

# Intermediate

Read about subqueries below and then complete the given problems using the psotgres sandbox. Save your answers in a new file called subqueries.sql, which you can create in the lab exercise folder you've been working in.

## Subqueries

Subqueries are a queries that rely on an outer query.

A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved.

## Example: Subqueries

Let's say we want to konw the last names of authors who have written a book that's over 1000 pages and we have the following tables:

| | |
|---|---|
| 1 | Hawkins |
| 2 | Gold |
| 3 | Williams |
| 4 | Anderson |

| | | | |
|---|---|---|---|
| 1 | Big Book | 2300 | 2 |
| 2 | Book | 400 | 4 |
| 3 | Tiny Book | 8 | 1 |
| 4 | The Book | 1500 | 3 |

## Syntax: Subqueries

We could write a subquery like this:

```
SELECT *
FROM authors
WHERE author_id IN (
  SELECT author_id FROM books WHERE pages > 1000
);
```

And get this result:

| | |
|---|---|
| 2 | Gold |
| 3 | Williams |

## Update Using a Subquery

What if you wanted to update the number of page's in "Tiny Book" but didn't know its book_id? Use a subquery!

```
UPDATE books
SET pages = 7
WHERE book_id IN (
  SELECT book_id
  FROM books
  WHERE title = 'Tiny Book'
);
```

## Delete Using a Subquery

We could do a similar thing to delete data.

```
DELETE
FROM books
WHERE author_id IN (
  SELECT author_id
  FROM authors
  WHERE author = 'Williams'
);
```

## Problems

Open up the postgres sandbox to complete these problems. Save your answers in a file subqueries.sql. Push to GitHub when you're done.

. Get all invoices where the ***unit_price*** on the ***invoice_line*** is greater than $0.99.

. Get all playlist tracks where the playlist name is Music.

. Get all track names for ***playlist_id*** 5.

. Get all tracks where the ***genre*** is Comedy.

. Get all tracks where the ***album*** is Fireball.

. Get all tracks for the artist Queen ( 2 nested subqueries ).