WEEK-10 DAY-3
Deeper Into Data

ORM Learning Objectives

To ease the use of SQL, the object-relational mapping tool was invented. This allows developers to focus on their application code and let a library generate all of the SQL for them. Depending on which developer you ask, this is a miracle
or a travesty. Either way, those developers use them because writing all of the SQL by hand is a chore that most software developers just don't want to do.

In this section, you will learn:

- How to install, configure, and use Sequelize, an ORM for JavaScript
- How to use database migrations to make your database grow with your application in a source-control enabled way
- How to perform CRUD operations with Sequelize
- How to query using Sequelize
- How to perform data validations with Sequelize
- How to use transactions with Sequelize

Creating A Schema For Relational Database Design

Schemas allow use to easily visualize database tables and their relationships to
one another, so that we can identify areas that need clarity, refinement, or redesign.

In this reading, we're going to cover the stages of relational database design and how to create schema that depicts database table relationships.

What is Relational Database Design?

According to Technopedia, Relational Database Design (or RDD) differs from other databases in terms of data organization and transactions: "In an RDD, the
data are organized into tables and all types of data access are carried out via controlled transactions."

In previous readings, we created relational database tables and accessed data
from these tables through PostgreSQL queries. These tables (a.k.a. *entities*) contain rows (a.k.a. *records*) and columns (a.k.a. *fields*). We also learned how to uniquely identify table records by adding a `PRIMARY KEY` and how to create a table association by adding a `FOREIGN KEY`.

A relational database usually contains multiple tables. It's useful to create schema to help us visualize these tables, keep track of primary keys and foreign
keys, and create relationships among tables. This is a key part of the RDD process defined below.

Stages of Relational Database Design

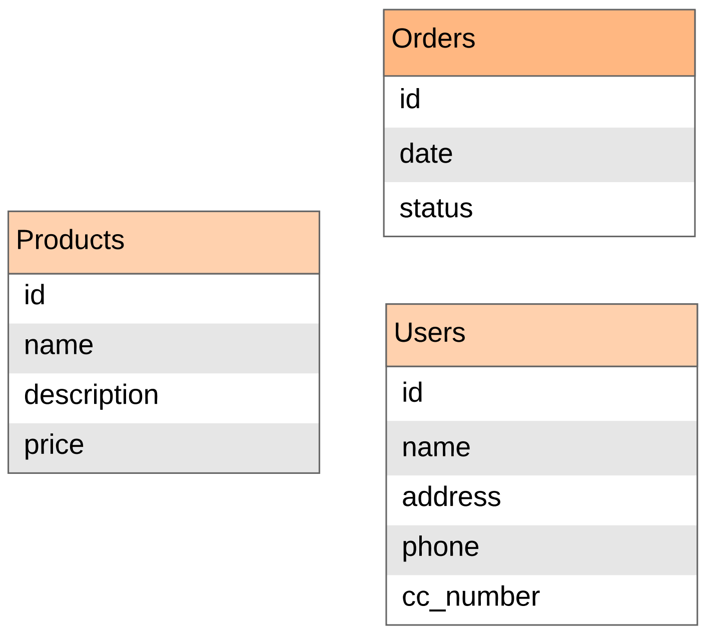
There are four generally-agreed-upon stages of Relational Database Design:

1. Define the purpose/entities of the relational DB.
2. Identify primary keys.
3. Establish table relationships.
4. Apply normalization rules.

1. Define database purpose and entities

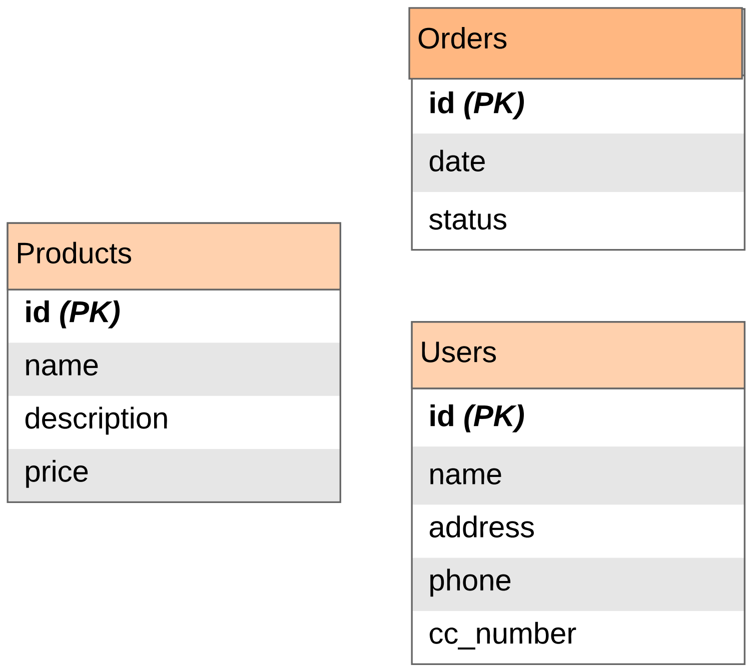
The first stage is identifying the purpose of the database (*Why is the database being created? What problem is it solving? What is the data used for?*), as well
as identifying the main entities, or *tables*, that need to be created. It also typically involves identifying the table's attributes (i.e. *columns* and
rows).

For example, if we were creating a database for order processing on an e-commerce application, we would need a database with at least three tables: a
`products` table, an `orders` tables, and a `users` (i.e. customers) table. We know that a product will probably have an ID, name, and price, and an order will
contain one or more product IDs. We also know that users can create multiple orders.



2. Identify primary keys

The second stage is to identify the primary key (*PK*) of each table. As we previously learned, a table’s primary key contains a unique value, or values, that identify each distinct record. For our above example of online orders, we would probably create IDs to serve as the primary key for each table: a product
ID, an order ID, and a user ID.



3. Establish table relationships

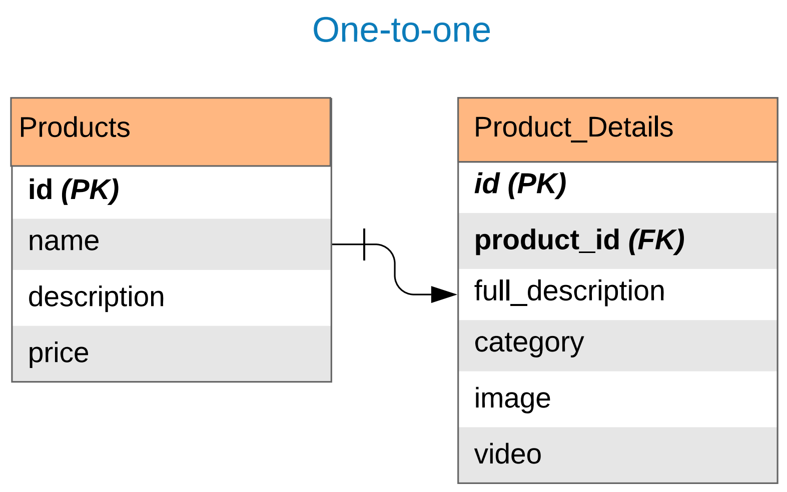
The third stage is to establish the relationships among the tables in the database. There are three types of relational database table relationships:

- One-to-one
- One-to-many
- Many-to-many

One-to-one relationship

In a one-to-one relationship, one record in a table is associated with only one record in another table. We could say that only one record in Table B belongs to
only one record in Table A.

A one-to-one relationship is the least common type of table relationship. While the two tables above could be combined into a single table, we may want to keep
some less-used data separate from the main `products` table.



The above schema depicts two tables: a “products” table and a “product_details” table. A `product_details` record belongs to only one product record. We've used
an arrow to indicate the one-to-one relationship between the tables. Both tables
have the same primary key -- `product_id` -- which we can use in a `JOIN`
operation to get data from both tables.

This table relationship would produce the following example data (note that not
all columns are shown below):

Products

id	name
1597	Glass Coffee Mug
1598	Metallic Coffee Mug
1599	Smart Coffee Mug

Product Details

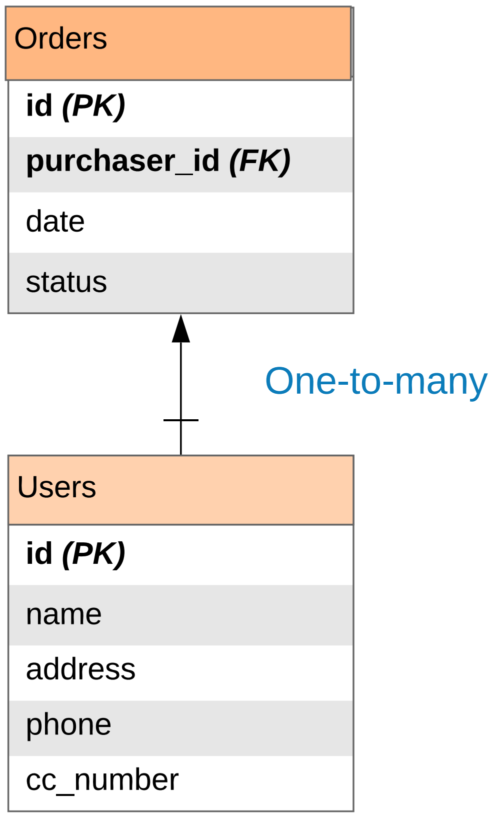
id	product_id	full_description
1	1597	Sturdy tempered glass coffee mug with natural cork band and silicone lid. Barista standard - fits under commercial coffee machine heads and most cup-holders.

id	product_id	full_description
2	1598	Fun coffee mug that comes in various metallic colors. Sleek, stylish, and easy to wash. Makes a great addition to your kitchen. Take it on the go by attaching the secure lid.
3	1599	This smart mug goes beyond being a simple coffee receptacle. Its smart features let you set and maintain an exact drinking temperature for up to 1.5 hours, so your coffee is never too hot or too cold.

Take a moment to analyze the data above. Using the foreign keys, you should be
able to reason out that the "Metallic Coffee Mug" is a "Fun coffee mug that comes in various metallic colors."

One-to-many relationship

In a one-to-many relationship, each record in Table A is associated with multiple records in Table B. Each record in Table B is associated with only one record in Table A.



The above schema depicts a one-to-many relationship between the “users” table
and the `orders` table: One user can create multiple orders. The primary key of
the “orders” table (`id`) is a foreign key in the “users” table (`order_id`). We can use this foreign key in a `JOIN` operation to get data from both tables.

This table relationship would produce the following example data (note that not
all columns are shown below):

Users

id	name
1	Alice
2	Bob

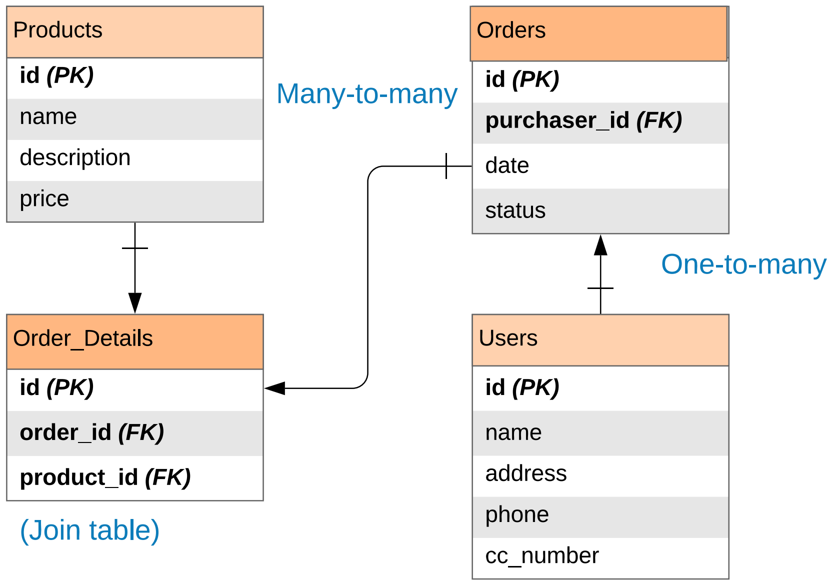
Orders

id	purchaser_id
10	1
11	1
12	2

Take a moment to analyze the data above. Using the foreign keys, you should be
able to reason out that "Alice" has made two orders and "Bob" has made one order.

Many-to-many relationship

In a many-to-many relationship, multiple records in Table A are associated with
multiple records in Table B. You would normally create a third table for this relationship called a ***join table***, which contains the primary keys from both tables.



The above schema depicts a many-to-many relationship between the `products`
table and the `orders` table. A single order can have multiple products, and a single product can belong to multiple orders. We created a third join table called `order_details`, which contains both the `order_id` and `product_id` fields as foreign keys.

This table relationship would produce the following example data(note that not all columns are shown below):

Products

id	name
1597	Glass Coffee Mug
1598	Metallic Coffee Mug
1599	Smart Coffee Mug

Users

id	name
1	Alice
2	Bob

Orders

id	purchaser_id
10	1
11	1
12	2

Order Details

id	order_id	product_id
1	10	1599
2	11	1597
3	11	1598
4	12	1597
5	12	1598
6	12	1599

Take a moment to analyze the data above. Using the foreign keys, you should be
able to reason out that "Alice" has two orders. One order containing a "Smart Coffee Mug" and another order containing both a "Glass Coffee Mug" and "Metallic
Coffee Mug".

4. Apply normalization rules

The fourth stage in RDD is ***normalization***. Normalization is the process of optimizing the database structure so that redundancy and confusion are eliminated.

The rules of normalization are called “normal forms” and are as follows:

1. First normal form
2. Second normal form
3. Third normal form
4. Boyce-Codd normal form
5. Fifth normal form

The first three forms are widely used in practice, while the fourth and fifth are less often used.

First normal form rules:

- Eliminate repeating groups in individual tables.
- Create a separate table for each set of related data.
- Identify each set of related data with a primary key.

Second normal form rules:

- Create separate tables for sets of values that apply to multiple records.
- Relate these tables with a foreign key.

Third normal form rules:

- Eliminate fields that do not depend on the table's key.

Note: For examples of how to apply these forms, read "Description of the database normalization basics" from Microsoft.

Schema design tools

Many people draw their relational database design schema with good ol' pen and
paper, or on a whiteboard. However, there are also lots of online tools created for this purpose if you'd like to use something easily exportable/shareable. Feel free to check out the ERD (short for "Entity Relationship Diagram") tools below.

Free Database Diagram (ERD) Design Tools:

- Lucidchart
- draw.io
- dbdiagram.io
- QuickDBD

What we learned:

- Stages of Relational Database Design (RDD)
- Examples of schema depicting table relationships
- Normalization rules
- Schema drawing tools

Using SQL Transactions

Transactions allow us to make changes to a SQL database in a consistent and
durable way, and it's a best practice to use them regularly.

In this reading, we'll cover what a transaction is and why we want to use it, as well as how to write explicit transactions.

What is a transaction?

A transaction is a single unit of work, which can contain multiple operations, performed on a database. According to the PostgreSQL docs, the important thing to note about a transaction is that "it bundles multiple steps into a single, all-or-nothing operation”. If any operation within the transaction fails, then the entire transaction fails. If all the operations succeed, then the entire transaction succeeds.

Implicit vs. explicit transactions

Every SQL statement is effectively a transaction. When you insert a new table row into a database table, for example, you are creating a transaction. The following `INSERT` statement is a transaction:

```
INSERT INTO hobbits(name,purpose)
  VALUES('Frodo','Destroy the One Ring of power.');
```

The above code is known as an *implicit* transaction. With an implicit transaction, changes to the database happen immediately, and we have no way to
undo or roll back these changes. We can only make subsequent changes/ transactions.

An *explicit* transaction, however, allows us to create save points and roll back to whatever point in time we choose. An explicit transaction begins with the command `BEGIN`, followed by the SQL statement, and then ends with either a
`COMMIT` or `ROLLBACK`.

PostgreSQL transactional commands

BEGIN

-- Initiates a transaction block. All statements after a BEGIN command will be executed in a single transaction until an explicit COMMIT or ROLLBACK is given.

Starting a transaction:

```
BEGIN;
  INSERT INTO hobbits(name,purpose)
    VALUES('Frodo','Destroy the One Ring of power.');
```

COMMIT

-- Commits the current transaction. All changes made by the transaction become
visible to others and are guaranteed to be durable if a crash occurs.

Committing a transaction:

```
BEGIN;
  INSERT INTO hobbits(name,purpose)
    VALUES('Frodo','Destroy the One Ring of power.');
COMMIT;
```

ROLLBACK

-- Rolls back the current transaction and causes all the updates made by the transaction to be discarded. Can only undo transactions since the last COMMIT or
ROLLBACK command was issued.

Rolling back a transaction (i.e. abort all changes):

```
BEGIN;
  INSERT INTO hobbits(name,purpose)
    VALUES('Frodo','Destroy the One Ring of power.');
ROLLBACK;
```

SAVEPOINT

-- Establishes a new save point within the current transaction. Allows all commands executed after the save point to be rolled back, restoring the transaction state to what it was at the time of the save point.

Syntax to create save point: `SAVEPOINT savepoint_name;`
Syntax to delete a save point: `RELEASE SAVEPOINT savepoint_name;`

Let's say we had the following table called `fellowship`:

name	age
Frodo	50
Samwise	38
Merry	36
Pippin	28
Aragorn	87
Boromir	40
Legolas	2000
Gandalf	2000

We'll create a transaction on this table containing a few operations. Inside the transaction, we'll establish a save point that we'll roll back to before committing.

```
BEGIN;
  DELETE FROM fellowship
    WHERE age > 100;
  SAVEPOINT first_savepoint;
  DELETE FROM fellowship
    WHERE age > 80;
  DELETE FROM fellowship
    WHERE age >= 40;
  ROLLBACK TO first_savepoint;
COMMIT;
```

Once our transaction is committed, our table would look like this:

name	age
Frodo	50
Samwise	38
Merry	36
Pippin	28
Aragorn	87
Boromir	40

We can see that the deletion that happened just prior to the savepoint creation
was preserved.

SET TRANSACTION
-- Sets the characteristics of the current transaction. (_Note: To set characteristics for subsequent transactions in a session, use `SET SESSION CHARACTERISTICS`.) The available transaction characteristics are the
transaction isolation level, the transaction access mode (read/write or read-only), and the deferrable mode. (*Read more about these characteristics in the PostgreSQL docs.*)

Example of setting transaction characteristics:

```
BEGIN;
  SET TRANSACTION READ ONLY;
  ...
COMMIT;
```

When to use transactions and why

It is generally a good idea to use explicit SQL transactions when making any updates, insertions, or deletions, to a database. However, you generally wouldn't write an explicit transaction for a simple `SELECT` query.

Transactions help you deal with crashes, failures, data consistency, and error handling. The ability to create savepoints and roll back to earlier points is tremendously helpful when doing multiple updates and helps maintain data integrity.

Another benefit of transactions is the ***atomic***, or "all-or-nothing", nature of their operations. Because all of the operations in a transaction must succeed or else be aborted, partial or incomplete updates to the database will not be made. End-users will see only the final result of the transaction.

Transaction properties: ACID

A SQL transaction has four properties known collectively as "ACID" -- which is an acronym for *Atomic, Consistent, Isolated, and Durable*. The following descriptions come from the IBM doc "ACID properties of transactions":

Atomicity
-- All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are.

For example, in an application that transfers funds from one account to another,
the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

Consistency
-- Data is in a consistent state when a transaction starts and when it ends.

For example, in an application that transfers funds from one account to another,
the consistency property ensures that the total value of funds in both the accounts is the same at the start and end of each transaction.

Isolation
-- The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialized.

For example, in an application that transfers funds from one account to another,
the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability
-- After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure.

For example, in an application that transfers funds from one account to another,
the durability property ensures that the changes made to each account will not
be reversed.

Banking transaction example

Let’s look at an example from the PostgreSQL Transactions doc that demonstrates the ACID properties of a transaction. We have a bank database that
contains customer account balances, as well as total deposit balances for branches. We want to record a payment of $100.00 from Alice's account to Bob's
account, as well as update the total branch balances. The transaction would look like the code below.

```
BEGIN;
  UPDATE accounts SET balance = balance - 100.00
      WHERE name = 'Alice';
  UPDATE branches SET balance = balance - 100.00
      WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Alice'
  UPDATE accounts SET balance = balance + 100.00
      WHERE name = 'Bob';
  UPDATE branches SET balance = balance + 100.00
      WHERE name = (SELECT branch_name FROM accounts WHERE name = 'Bob');
COMMIT;
```

There are several updates happening above. The bank wants to make sure that all
of the updates happen or none happen, in order to ensure that funds are transferred from the proper account (i.e. Alice’s account) to the proper recipient’s account (i.e. Bob’s account). If any of the updates fails, none of them will take effect. That is, if something goes wrong either with withdrawing funds from Alice’s account or transferring the funds into Bob’s account, then the entire transaction will be aborted and no changes will occur. This prevents Alice or Bob from seeing a transaction in their account summaries that isn’t supposed to be there.

There are many other scenarios where we would want to use an atomic operation to
ensure a successful end result. Transactions are ideal for such scenarios, and we should use them whenever they’re applicable.

Helpful links:

- PostgreSQL: Transactions

- PostgreSQL Tutorial: PostgreSQL Transaction
- PostgreSQL: BEGIN
- PostgreSQL: COMMIT
- PostgreSQL: ROLLBACK
- PostgreSQL: SAVEPOINT
- PostgreSQL: SET TRANSACTION

Joins vs. Subqueries

To select, or not to select? That is the query. We've barely scratched the surface of SQL queries. Previously, we went over how to write simple SQL queries using the `SELECT` statement, and we learned how to incorporate a `WHERE` clause into our queries.

There's a lot more we could add to our queries to get more refined results. In this reading, we'll go over joins and subqueries and talk about when we would use one over the other.

What is a JOIN?

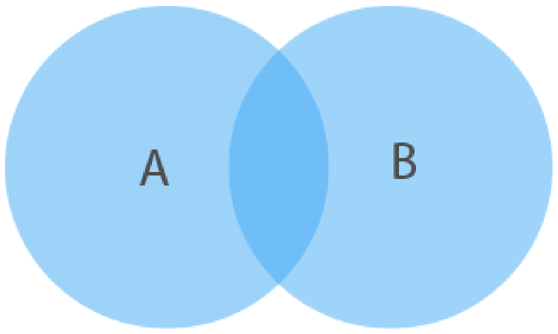
We briefly looked at the `JOIN` operation after we created foreign keys in a previous reading. The JOIN operation allows us to retrieve rows from multiple tables.

To review, we had two tables: a "breeds" table and a "puppies" table. The two tables shared information through a foreign key. The foreign key `breed_id` lives on the "puppies" table and is related to the primary key `id` of the "breeds" table.

We wrote the following `INNER JOIN` operation to get only the rows from the "puppies" table with a matching `breed_id` in the "friends" table:

```
SELECT * FROM puppies
INNER JOIN breeds ON (puppies.breed_id = breeds.id);
```

INNER JOIN can be represented as a Venn Diagram, which produces rows from
Table A that match some information in Table B.



PostgreSQL INNER JOIN

We got the following table rows back from our `INNER JOIN` on the "puppies" table. These rows represent the center overlapping area of the two circles. We
can see that the data from "puppies" appears first, followed by the joined data from the "friends" table.

```
 id |   name    | age_yrs | breed_id | weight_lbs | microchipped | id |
----+-----------+---------+----------+------------+--------------+----+--
  1 | Cooper    |     1.0 |        8 |         18 | t            |  8 | Mi
  2 | Indie     |     0.5 |        9 |         13 | t            |  9 | Yo
  3 | Kota      |     0.7 |        1 |         26 | f            |  1 | Au
  4 | Zoe       |     0.8 |        6 |         32 | t            |  6 | Ko
  5 | Charley   |     1.5 |        2 |         25 | f            |  2 | Ba
  6 | Ladybird  |     0.6 |        7 |         20 | t            |  7 | La
  7 | Callie    |     0.9 |        4 |         16 | f            |  4 | Co
  8 | Jaxson    |     0.4 |        3 |         19 | t            |  3 | Be
  9 | Leinni    |     1.0 |        8 |         25 | t            |  8 | Mi
 10 | Max       |     1.6 |        5 |         65 | f            |  5 | Ge
```

There are different types of `JOIN` operations. The ones you'll use most often are:

1. **Inner Join** -- Returns a result set containing rows in the left table that match rows in the right table.
2. **Left Join** -- Returns a result set containing all rows from the left table with the matching rows from the right table. If there is no match, the right side will have null values.
3. **Right Join** -- Returns a result set containing all rows from the right table with matching rows from the left table. If there is no match, the left side will have null values.
4. **Full Outer Join** -- Returns a result set containing all rows from both the left and right tables, with the matching rows from both sides where available. If there is no match, the missing side contains null values.
5. **Self-Join** -- A self-join is a query in which a table is joined to itself. Self-joins are useful for comparing values in a column of rows within the same table.

(See this tutorial doc on PostgreSQL Joins for more information on the different `JOIN` operations.)

What is a subquery?

A subquery is essentially a `SELECT` statement nested inside another `SELECT` statement. A subquery can return a single (“scalar”) value or multiple rows.

Single-value subquery

Let’s see an example of how to use a subquery to return a single value. Take the "puppies" table from before. We had a column called `age_yrs` in that table (see below).

```
postgres=# SELECT * FROM puppies;
 id |   name    | age_yrs | breed_id | weight_lbs | microchipped
----+-----------+---------+----------+------------+--------------

  1 | Cooper    |     1.0 |        8 |         18 | t
  2 | Indie     |     0.5 |        9 |         13 | t
  3 | Kota      |     0.7 |        1 |         26 | f
  4 | Zoe       |     0.8 |        6 |         32 | t
  5 | Charley   |     1.5 |        2 |         25 | f
  6 | Ladybird  |     0.6 |        7 |         20 | t
  7 | Callie    |     0.9 |        4 |         16 | f
  8 | Jaxson    |     0.4 |        3 |         19 | t
  9 | Leinni    |     1.0 |        8 |         25 | t
 10 | Max       |     1.6 |        5 |         65 | f
(10 rows)
```

We’ll use the PostgreSQL aggregate function `AVG` to get an average puppy age.

```
SELECT
  AVG (age_yrs)
FROM
  puppies;
```

Assuming our previous “puppies” table still exists in our database, if we entered the above statement into psql we’d get an ***average age of 0.9***. (*Note: Try it out yourself in psql! Refer to the reading "Retrieving Rows From A Table Using SELECT" if you need help remembering how we set up the “puppies”*
table.)

Let's say that we wanted to find all of the puppies that are older than the average age of 0.9. We could write the following query:

```
SELECT
  name,
  age_yrs,
  breed
FROM
  puppies
WHERE
  age_yrs > 0.9;
```

In the above query, we compared `age_yrs` to an actual number (0.9). However, as
more puppies get added to our table, the average age could change at any time.
To make our statement more dynamic, we can plug in the query we wrote to find
the average age into another statement as a *subquery* (surrounded by parentheses).

```
SELECT
  puppies.name,
  age_yrs,
  breeds.name
FROM
  puppies
INNER JOIN
  breeds ON (breeds.id = puppies.breed_id)
WHERE
  age_yrs > (
    SELECT
      AVG (age_yrs)
    FROM
      puppies
  );
```

We should get the following table rows, which include only the puppies older than 9 months:

```
  name   | age_yrs |        breed
---------+---------+---------------------
 Cooper  |     1.0 | Miniature Schnauzer
 Charley |     1.5 | Basset Hound
 Leinni  |     1.0 | Miniature Schnauzer
 Max     |     1.6 | German Shepherd
```

Multiple-row subquery

We could also write a subquery that returns multiple rows.

In the reading "Creating A Table In An Existing PostgreSQL Database", we created a "friends" table. In "Foreign Keys And The JOIN Operation", we set up a
primary key in the "puppies" table that is a foreign key in the "friends" table
-- `puppy_id`. We'll use this ID in our subquery and outer query.

"friends" table

```
id | first_name | last_name | puppy_id
----+------------+-----------+----------
  1 | Amy        | Pond      |        4
  2 | Rose       | Tyler     |        5
  3 | Martha     | Jones     |        6
  4 | Donna      | Noble     |        7
  5 | River      | Song      |        8
```

Let's say we wanted to find all the puppies that are younger than 6 months old.

```
SELECT puppy_id
FROM puppies
WHERE
  age_yrs < 0.6;
```

This would return two rows:

```
puppy_id
----------
        2
        8
(2 rows)
```

Now we want to use the above statement as a subquery (inside parentheses) in
another query. You'll notice we're using a `WHERE` clause with the IN
operator to check if the `puppy_id` from the "friends" table meets the conditions in the subquery.

```
SELECT *
FROM friends
WHERE
  puppy_id IN (
    SELECT puppy_id
    FROM puppies
    WHERE
      age_yrs < 0.6
  );
```

We should get just one row back. River Song has a puppy younger than 6 months
old.

```
 id | first_name | last_name | puppy_id
----+------------+-----------+----------
  5 | River      | Song      |        8
(1 row)
```

Using joins vs. subqueries

We can use either a `JOIN` operation or a subquery to filter for the same information. Both methods can be used to get info from multiple tables. Take the query/subquery from above:

```
SELECT *
FROM friends
WHERE
  puppy_id IN  (
    SELECT puppy_id
    FROM puppies
    WHERE
      age_yrs < 0.6
  );
```

Which produced the following result:

```
 id | first_name | last_name | puppy_id
----+------------+-----------+----------
  5 | River      | Song      |        8
(1 row)
```

Instead of using a `WHERE` clause with a subquery, we could use a `JOIN` operation:

```
SELECT *
FROM friends
INNER JOIN puppies ON (puppies.puppy_id = friends.puppy_id)
WHERE
  puppies.age_yrs < 0.6;
```

Again, we'd get back one result, but because we used an `INNER JOIN` we have
information from both the "puppies" and "friends" tables.

```
id | first_name | last_name | puppy_id |  name   | age_yrs | breed   | weig
----+------------+-----------+----------+--------+---------+--------+----
  5 | River      | Song      |        8 | Jaxson |     0.4 | Beagle |
(1 row)
```

Should I use a JOIN or a subquery?

As stated earlier, we could use either a JOIN operation or a subquery to filter for table rows. However, you might want to think about whether using a JOIN or a
subquery is more appropriate for retrieving data.

A JOIN operation is ideal when you want to combine rows from one or more tables
based on a match condition. Subqueries work great when you're returning a single
value. When you're returning multiple rows, you could opt for a subquery or a JOIN.

Executing a query using a JOIN could potentially be faster than executing a subquery that would return the same data. (A subquery will execute once for each
row returned in the outer query, whereas the `INNER JOIN` only has to make one
pass through the data.) However, this isn't always the case. Performance depends on the size of your data, what you're filtering for, and how the server optimizes the query. With smaller datasets, the difference in performance of a JOIN and subquery is imperceptible. However, there are use cases where a subquery would improve performance.

(*See this article for more info: When is a SQL Subquery 260x Faster than a Left Join?*)

We can use the the EXPLAIN statement to see runtime statistics of our queries that help with debugging slow queries. (We'll get into this more later!)

Helpful links:

- PostgreSQL Tutorial: PostgreSQL Joins
- PostgreSQL Tutorial: PostgreSQL Subquery
- PostgreSQL Docs: Subquery Expressions
- Essential SQL: Subqueries versus Joins
- Essential SQL: Using Subqueries with the Select Statement

PostgreSQL Indexes

PostgreSQL indexes can help us optimize our queries for faster performance. In
this reading, we'll learn how to create an index, when to use an index, and when
to avoid using them.

What is a PostgreSQL index?

A PostgreSQL index works like an index in the back of a book. It points to information contained elsewhere and can be a faster method of looking up the information we want.

A book index contains a list of references with page numbers. Instead of having
to scan all the pages of the book to find the places where specific information appears, a reader can simply check the index. In similar fashion, PostgreSQL indexes, which are special lookup tables, let us make faster database queries.

Let’s say we had the following table:

addresses
address_id
address
address2
city_id
postal_code
phone_number

And we made a query to the database like the following:

```
SELECT * FROM addresses WHERE phone_number = '5556667777';
```

The above query would scan every row of the "addresses" table to find matching
rows based on the given phone number. If "addresses" is a large table (let's say
with millions of entries), and we only expect to get a small number of results

back (one row, or a few rows), then such a query would be an inefficient way
to
retrieve data. Instead of scanning every row, we could create an index for the
phone column for faster data retrieval.

How to create an index

To create a PostgreSQL index, use the following syntax:

```
CREATE INDEX index_name ON table_name (column_name);
```

We can create a phone number index for the above "addresses" table with the following:

```
CREATE INDEX addresses_phone_index ON addresses (phone_number);
```

You can delete an index using the `DROP INDEX` command:

```
DROP INDEX addresses_phone_index;
```

After an index has been created, the system will take care of the rest -- it
will update an index when the table is modified and use the index in queries
when it improves performance over a full table scan.

Types of indexes

PostgreSQL provides several index types: B-tree, Hash, GiST and GIN. The
CREATE
INDEX command creates B-tree indexes by default, which fit the most
common
situations. While it's good to know other index types exist, you'll probably
find yourself using the default B-tree most often.

Single-Column Indexes
Uses only one table column.

Syntax:

```
CREATE INDEX index_name ON table_name (column_name);
```

Addresses Example:

```
CREATE INDEX addresses_phone_index ON addresses (phone_number);
```

Multiple-Column Indexes
Uses more than one table column.

Syntax:

```
CREATE INDEX index_name ON table_name (col1_name, col2_name);
```

Addresses Example:

```
CREATE INDEX idx_addresses_city_post_code ON table_name (city_id, postal_
```

Partial Indexes
Uses subset of a table defined by a conditional expression.

Syntax:

```
CREATE INDEX index_name ON table_name WHERE (conditional_expression);
```

Addresses Example:

```
CREATE INDEX addresses_phone_index ON addresses (phone_number) WHERE (cit
```

Note: Check out Chapter 11 on Indexes in the PostgreSQL docs for more about types of indexes.

When to use an index

Indexes are intended to enhance database performance and are generally thought
to be a faster data retrieval method than a sequential (or row-by-row) table scan. However, there are instances where using an index would not improve efficiency, such as the following:

- When working with a relatively small table (i.e. not a lot of rows)
- When a table has frequent, large-batch update or insert operations
- When working with columns that contain many NULL values
- When working with columns that are frequently manipulated

An important thing to note about indexes is that, while they can optimize READ
(i.e. table query) speed, they can also hamper WRITE (i.e. table updates/insertions) speed. The latter's performance is affected due to the system having to spend time updating indexes whenever a change or insertion is
made to the table.

The system optimizes database performance and decides whether to use an index in a query or to perform a sequential table scan, but we can analyze query
performance ourselves to debug slow queries using `EXPLAIN` and `EXPLAIN ANALYZE`.

Here is an example of using `EXPLAIN` from the PostgreSQL docs:

```
EXPLAIN SELECT * FROM tenk1;

                             QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

In the QUERY PLAN above, we can see that a sequential table scan (`Seq Scan`)
was performed on the table called "tenk1". In parentheses, we see performance
information:

- Estimated start-up cost (or time expended before the scan can start): 0.00
- Estimated total cost: 458.00
- Estimated number of rows output: 10000
- Estimated average width (in bytes) of rows: 244

It's important to note that, although we might mistake the number next to `cost` for milliseconds, `cost` is not measured in any particular unit and is an arbitrary measurement relatively based on other query costs.

If we use the `ANALYZE` keyword after `EXPLAIN` on a `SELECT` statement, we can get more information about query performance:

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                                             QUERY PLAN
--------------------------------------------------------------------------
 Nested Loop  (cost=2.37..553.11 rows=106 width=488) (actual time=1.392..
   ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=24
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=10
               Index Cond: (unique1 < 100)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=
         Index Cond: (unique2 = t1.unique2)
 Total runtime: 14.452 ms
```

We can see in the QUERY PLAN above that there are other types of scans happening: Bitmap Heap Scan, Bitmap Index Scan, and Index Scan. We know that an
index has been created, and the system is using it to scan for results instead of performing a sequential scan. At the bottom, we also have a total runtime of
14.42 ms, which is pretty fast.

Helpful links:

- PostgreSQL docs: Indexes
- PostgreSQL docs: Performance Tips
- Heroku DevCenter: Efficient Use of PostgreSQL Indexes

What we learned:

- How to create a PostgreSQL index
- Types of indexes
- Use cases for indexes and when to avoid them
- How to use `EXPLAIN` to analyze query performance

Node-Postgres And Prepared Statements

The library node-postgres is, not too surprisingly, the library that Node.js applications use to connect to a database managed by a PostgreSQL RDBMS. The
applications that you deal with will use this library to make connections to the database and get rows returned from your SQL `SELECT` statements.

Connecting

The **node-postgres** library provides two ways to connect to it. You can use a single `Client` object, or you can use a `Pool` of `Client` objects. Normally, you want to use a `Pool` because creating and opening single connections to

any
database server is a "costly" operation in terms of CPU and network resources.
A `Pool` creates a group of those `Client` connections and lets your code use them whenever it needs to.

To use a `Pool`, you specify any specific portions of the following connection parameters that you need. The default values for each parameter is listed with each parameter.

Connection parameter	What it indicates	Default value
user	The name of the user you want to connect as	The user name that runs the application
password	The password to use	The password set in your configuration
database	The name of the database to connect to	The user's database
port	The port over which to connect	5432
host	The name of the server to connect to	localhost

Normally, you will only override the user and password fields, and sometimes the
database if it doesn't match the user's name. You do that by instantiating a new `Pool` object and passing in an object with those key/value pairs.

```
const { Pool } = require('pg');

const pool = new Pool({
  user: 'application_user',
  password: 'iy7qTEcZ',
});
```

Once you have an instance of the `Pool` class, you can use the `query` method on
it to run queries. (The `query` method returns a Promise, so it's nice to just use `await` for it to finish.)

```
const results = await pool.query(`
  SELECT id, name, age_yrs
  FROM puppies;
`);

console.log(results);
```

When this runs, you will get an object that contains a property named "rows". The value of "rows" will be an array of the rows that match the statement. Here's an example output from the code above.

```
{
  rows:
    [ { id: 1, name: 'Cooper', age_yrs: '1.0' },
      { id: 2, name: 'Indie', age_yrs: '0.5' },
      { id: 3, name: 'Kota', age_yrs: '0.7' },
      { id: 4, name: 'Zoe', age_yrs: '0.8' },
      { id: 5, name: 'Charley', age_yrs: '1.5' },
      { id: 6, name: 'Ladybird', age_yrs: '0.6' },
      { id: 7, name: 'Callie', age_yrs: '0.9' },
      { id: 8, name: 'Jaxson', age_yrs: '0.4' },
      { id: 9, name: 'Leinni', age_yrs: '1.0' },
      { id: 10, name: 'Max', age_yrs: '1.6' } ],
}
```

You can see that the "rows" property contains an array of objects. Each object represents a row in the "puppies" table that matches the query. Each object has
a property named after the column selected in the `SELECT` statement. The query
read `SELECT id, name, age_yrs` and each object has an "id", "name", and an "age_yrs" property.

You can then use that array to loop over and do things with it. For example, you could print them to the console like this:

```
for (let row of results.rows) {
  console.log(`${row.id}: ${row.name} is ${row.age_yrs} old</li>`);
}
```

Which would show

```
1. Cooper is 1.0 years old
2. Indie is 0.5 years old
3. Kots is 0.7 years old
...
```

Prepared Statement

Prepared statements are SQL statements that have parameters in them that you can
use to substitute values. The parameters are normally part of the `WHERE` clause
in all statements. You will also use then in the `SET` portion of `UPDATE` statements and the `VALUES` portion of `INSERT` statements.

For example, say your application collected the information to create a new row
in the puppy table by asking for the puppy's name, age, breed, weight, and if it was microchipped. You'd have those values stored in variables somewhere. You'd want to create an `INSERT` statement that inserts the data from those variables into a SQL statement that the RDBMS could then execute to insert a new row.

Think about what a generic `INSERT` statement would look like. It would have to
have the

```
INSERT INTO puppies (name, age_yrs, breed, weight_lbs, microchipped)
```

portion of the statement. The part that would change with each time you
inserted
would be the specific values that would go into the `VALUES` section of the `INSERT` statement. With prepared statements, you use *positional parameters* to act as placeholders for the actual values that you will provide the query.

For example, the generic `INSERT` statement from above would look like this.

```
INSERT INTO puppies (name, age_yrs, breed, weight_lbs, microchipped)
VALUES ($1, $2, $3, $4, $5);
```

Each of the "
"*placeholdersisapositionalargumentfortheparametersthatyoupa*
1" placeholder is, which is the value for the "name" of the
puppy. The "$2" corresponds to the "age_yrs" column, so it should contain the age of the puppy. This continues for the third, fourth, and fifth parameters, as well.

Assume that in your code, you have the variables `name`, `age`, `breedName`, `weight`, and `isMicrochipped` containing the values that the user provided for the new puppy. Then, your use of the `query` method will now include another argument, the values that you want to pass in inside an array.

```
await pool.query(`
  INSERT INTO puppies (name, age_yrs, breed, weight_lbs, microchipped)
  VALUES ($1, $2, $3, $4, $5);
`, [name, age, breedName, weight, isMicrochipped]);
```

You can see that the variable `name` is in the first position of the array, so
it will be substituted into the placeholder "
1".*The‘age‘variableisinthesecondpositionofthearray,soitwillbes*
2"
placeholder.

The full documentation for how to use queries with **node-postgres** can be found on the Queries documentation page on their Web site.

Try it out for yourself

Make sure you have a database with a table that has data in it. If you don't, create a new database and run the following SQL.

```
CREATE TABLE puppies (
  id SERIAL PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  age_yrs NUMERIC(3,1) NOT NULL,
  breed VARCHAR(100) NOT NULL,
  weight_lbs INTEGER NOT NULL,
  microchipped BOOLEAN NOT NULL DEFAULT FALSE
);

insert into puppies(name, age_yrs, breed, weight_lbs, microchipped)
values

('Cooper', 1, 'Miniature Schnauzer', 18, 'yes'),
('Indie', 0.5, 'Yorkshire Terrier', 13, 'yes'),
('Kota', 0.7, 'Australian Shepherd', 26, 'no'),
('Zoe', 0.8, 'Korean Jindo', 32, 'yes'),
('Charley', 1.5, 'Basset Hound', 25, 'no'),
('Ladybird', 0.6, 'Labradoodle', 20, 'yes'),
('Callie', 0.9, 'Corgi', 16, 'no'),
('Jaxson', 0.4, 'Beagle', 19, 'yes'),
('Leinni', 1, 'Miniature Schnauzer', 25, 'yes' ),
('Max', 1.6, 'German Shepherd', 65, 'no');
```

Now that you have ten rows in the "puppies" table of a database, you can create
a simple Node.js project to access it.

Create a new directory somewhere that's not part of an existing project. Run `npm init -y` to initialize the **package.json** file. Then, run `npm install pg` to install the library from this section. (Why is the name of the library "node-postgres" but you install "pg"? Dunno.) Finally, open Visual Studio Code
for the current directory with `code .`.

Create a new file named **sql-test.js**.

The first thing you need to do is import the `Pool` class from the **node-postgres** library. The name of the library in the **node_modules** directory is "pg". That line of code looks like this and can be found all over the node-postgres documentation.

```
const { Pool } = require('pg');
```

Now, write some SQL that will select all of the records from the "puppies" table. (This is assuming you want to select puppies. If you're using a different table with different data, write the appropriate SQL here.)

```
const { Pool } = require('pg');

const allPuppies = `
  SELECT id, name, age_yrs, breed, weight_lbs, microchipped
  FROM puppies;
`;
```

You will now use that with a new `Pool` object. You will need to know the name
of the database that the "puppies" table is in (or whatever database you want to connect to).

```
const { Pool } = require('pg');

const allPuppies = `
  SELECT id, name, age_yrs, breed, weight_lbs, microchipped
  FROM puppies;
`;

const pool = new Pool({
  database: '«database name»'
});
```

Of course, replace "«database name»" with the name of your database. Otherwise,
when you run it, you will see this error message.

```
UnhandledPromiseRejectionWarning: error: database "«database name»" does not exist
```

This will, by default, connect to "localhost" on port "5432" with your user credentials because you did not specify any other parameters.

Once you have the pool, you can execute the query that you have in `allPuppies`.
Remember that the `query` method returns a Promise. This code wraps the call to
`query` in an `async function` so that it can use the `await` keyword for simplicity's sake. Then, it prints out the rows that it fetched to the console.

Finally, it calls `end` on the connection pool object to tell **node-postgres** to close all the connections and quit. Otherwise, your application will just hang and you'll have to close it with Control+C.

```
const { Pool } = require('pg');

const allPuppiesSql = `
  SELECT id, name, age_yrs, breed, weight_lbs, microchipped
  FROM puppies;
`;

const pool = new Pool({
  database: '«database name»'
});

async function selectAllPuppies() {
  const results = await pool.query(allPuppiesSql);
  console.log(results.rows);
  pool.end();
}

selectAllPuppies();
```

When you run this with `node sql-test.js`, you should see some output like this
although likely in a nicer format.

```
[ { id: 1, name: 'Cooper', age_yrs: '1.0', breed: 'Miniature Schnauzer', weight_lbs: 
  { id: 2, name: 'Indie', age_yrs: '0.5', breed: 'Yorkshire Terrier', weight_lbs: 13,
  { id: 3, name: 'Kota', age_yrs: '0.7', breed: 'Australian Shepherd', weight_lbs: 26
  { id: 4, name: 'Zoe', age_yrs: '0.8', breed: 'Korean Jindo', weight_lbs: 32, microc
  { id: 5, name: 'Charley', age_yrs: '1.5', breed: 'Basset Hound', weight_lbs: 25, mi
  { id: 6, name: 'Ladybird', age_yrs: '0.6', breed: 'Labradoodle', weight_lbs: 20, mi
  { id: 7, name: 'Callie', age_yrs: '0.9', breed: 'Corgi', weight_lbs: 16, microchipp
  { id: 8, name: 'Jaxson', age_yrs: '0.4', breed: 'Beagle', weight_lbs: 19, microchip
  { id: 9, name: 'Leinni', age_yrs: '1.0', breed: 'Miniature Schnauzer', weight_lbs: 
  { id: 10, name: 'Max', age_yrs: '1.6', breed: 'German Shepherd', weight_lbs: 65, mi
```

Now, try one of those parameterized queries. Comment out the
`selectAllPuppies`
function and invocation.

```
// async function selectAllPuppies() {
//   const results = await pool.query(allPuppiesSql);
//   console.log(results.rows);
//   pool.end();

// }

// selectAllPuppies();
```

Add the following content to the bottom of the file.

```
// Define the parameterized query where it will select a puppy
// based on an id
const singlePuppySql = `
  SELECT id, name, age_yrs, breed, weight_lbs, microchipped
  FROM puppies
  WHERE ID = $1;
`;

// Run the parameterized SQL by passing in an array that contains
// the puppyId to the query method. Then, print the results and
// end the pool.
async function selectOnePuppy(puppyId) {
  const results = await pool.query(singlePuppySql, [puppyId]);
  console.log(results.rows);
  pool.end();
}

// Get the id from the command line and store it
// in the variable "id". Pass that value to the
// selectOnePuppy function.
const id = Number.parseInt(process.argv[2]);
selectOnePuppy(id);
```

Now, when you run the program, include a number after the command. For example,
if you run `node sql-test.js 1`, it will print out

```
[ { id: 1,
    name: 'Cooper',
    age_yrs: '1.0',
    breed: 'Miniature Schnauzer',
    weight_lbs: 18,
    microchipped: true } ]
```

If you run `node sql-test.js 4`, it will print out

```
[ { id: 4,
    name: 'Zoe',
    age_yrs: '0.8',
    breed: 'Korean Jindo',
    weight_lbs: 32,
    microchipped: true } ]
```

That's because the number that you type on the command line is being substituted
in for the "$1" in the parameterized query. That means, when you pass in "4", It's like the RDMBS takes the parameterized query

```
SELECT id, name, age_yrs, breed, weight_lbs, microchipped
FROM puppies
WHERE ID = $1;
```

and your value "4"

and mushes them together to make

```
SELECT id, name, age_yrs, breed, weight_lbs, microchipped
FROM puppies
WHERE ID = 4; -- Value substituted here by PostgreSQL.
```

That happens because when you run the query, you call the `query` method like
this.

```
await pool.query(singlePuppySql, [puppyId]);
```

which passes along the sql stored in `singlePuppySql` and the value stored in `puppyId` (as the first parameter) to PostgreSQL.

What do you think will happen if you change `singlePuppySql` to have *two* parameters instead of one, but only pass in one parameter through the `query` method?

```
const singlePuppySql = `
  SELECT id, name, age_yrs, breed, weight_lbs, microchipped
  FROM puppies
  WHERE ID = $1
  AND age_yrs > $2;
`;
```

PostgreSQL is smart enough to see that you've only provided one parameter value
but it needs *two* positional parameters. It gives you the error message

```
error: bind message supplies 1 parameters, but prepared statement "" requires 2
```

In this error message, the term "bind message" is the kind of message that the
`query` method sends to PostgreSQL when you provide parameters to it.

Change the query back to how it was. Now, add an extra parameter to the invocation of the `query` method. What do you think will

```
await pool.query(singlePuppySql, [puppyId, 'extra parameter']);
```

Again, PostgreSQL gives you an error message about a mismatch in the number of placeholders in the SQL and the number of values you passed to it.

```
error: bind message supplies 2 parameters, but prepared statement "" requires 1
```

What you've learned

Here, you've seen how to connect to a PostgreSQL database using the **node-postgres** library named "pg". You saw how to run simple SQL statements
against it and handle the results. You also saw how to create parameterized queries so that you can pass in values to be substituted.

If you are using the **node-postgres** library and running your own handcrafted SQL, you will most often use parameterized queries. It's good to get familiar with them.

Transactions With Sequelize

In this reading, we will learn about database *transactions* and how we use them via Sequelize. We will learn how to group multiple update operations into a single atomic, indivisible unit.

At the end of the reading, you should know:

1. How to write code that is resilient to SQL operation failures,
2. How to group multiple operations into a database transaction using Sequelize (the `sequelize.transaction` method),

3. How to prevent "race conditions" using transactions.

The Problem: Database Updates Can Fail

Imagine a scenario with a banking database. Markov wants to transfer $7,500 to Curie. To perform the transfer, we will perform two database update operations:

1. Reduce Markov's account balance by $7,500,
2. Increase Curie's account balance by $7,500.

When transferring money, it's very important that *both* operations be performed. If we reduce Markov's balance but fail to increase Curie's balance, the bank effectively steals money from Markov. If we increase Curie's balance without reducing Markov's balance, the bank effectively gives away free money to Curie. Neither is acceptable.

We must keep in mind that any attempt to perform a database update can sometimes *fail*. It can fail for a number of reasons:

1. The command is sent, but the database has previously been shut down by the database administrator. Because the database is not running, the database is not listening for our update, won't receive it, and thus won't process it.
2. A bug in the database or operating system software has caused the database or operating system to crash. Again, the database is not running, so it can neither receive nor process our update.
3. Power has been lost to the machine running the database. The database is not running.
4. The internet connection that connects us to the database machine is disrupted. The database may be running and listening for SQL requests to process. However, our update request cannot get through to the database machine. Because the database cannot receive our request, the database cannot process it.
5. The update asks the database to violate a pre-defined constraint. For example: the database may have a constraint that an account balance must never be less than zero. Any update that asks the database to reduce an account balance to less than zero will be rejected and therefore fail.

Only this last scenario is "our fault." The fact is that database updates can fail **through no fault of our own**. With regard to our example: our first SQL request to reduce Markov's account balance may succeed, but the database may then crash before we have sent the request to increase Curie's balance. Through no fault of our own, the bank has stolen money from Markov without giving it to Curie.

How can we write code that avoids this fundamental problem?

The Solution: Database Transactions

One way to solve the problem is to "group" or "pair" the two update operations somehow. We want to tell the database "Reduce Markov's balance AND increment Curie's balance." We want to tell the database: "If for any reason you cannot perform **both** operations, make sure not to perform **either**." We want to tell the database: "If you crash after reducing Markov's balance, make sure to either (a) increase

Curie's balance when you restart, or (b) undo the increase to Markov's balance when you restart."

We want to ask the database to treat the pair of updates as one *atomic* (meaning **indivisible**) unit. SQL lets you do this using a feature called *transactions*.

You've previously seen how to use SQL transactions:

```sql
START TRANSACTION;
-- Reduce Markov's balance by $7500
UPDATE "BankAccounts" SET balance = balance - 7500 WHERE id = 1;
-- Increment Curie's balance by $7500
UPDATE "BankAccounts" SET balance = balance + 7500 WHERE id = 2;
COMMIT TRANSACTION;
```

SQL guarantees to you that everything between `START TRANSACTION` and `COMMIT TRANSACTION` will be processed *atomically*. If any update operation fails, none of the updates will be performed. The transaction is "all-or-nothing."

In this reading you will learn how to use SQL transactions with the Sequelize ORM.

The `BankAccounts` Schema

For our example in this reading, I will use a single table with two accounts.

```
catsdb=> SELECT * FROM "BankAccounts";
 id | clientName | balance | ...
----+------------+---------+-----
  1 | Markov     |    5000 | ...
  2 | Curie      |   10000 | ...
(2 rows)
```

I have generated a Sequelize model corresponding to the `BankAccounts` table:

```
// ./models/bank_account.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  // Define BankAccount model.
  const BankAccount = sequelize.define('BankAccount', {
    // Define clientName attribute.
    clientName: {
      type: DataTypes.STRING,
      allowNull: false,
      // Define validations on clientName.
      validate: {
        // clientName must not be null.
        notNull: {
          msg: "clientName must not be NULL",
        },
        // clientName must not be empty.
        notEmpty: {
          msg: "clientName must not be empty",
        },
      },
    },

    // Define balance attribute.
    balance: {
      type: DataTypes.INTEGER,
      allowNull: false,
      // Define validations on balance.
      validate: {
        // balance must not be less than zero.
        min: {
          args: [0],
          msg: "balance must not be less than zero",
        },
      },
    },
  }, {});

  return BankAccount;
};
```

Notice that the `min` validation on `balance` will not allow us to save an account balance that is below zero.

Example: An Update Fails Because Of Validation Failure

Let's imagine that Markov wants to transfer $7,500 to Curie. Unfortunately, Markov has only$5,000 in his account! Decrementing
Markov's account balance by $7,500 would put it in the negative, which our validation will not allow. Thus the transfer must fail.

Imagine that Markov is unaware that his account balance cannot cover the transfer. He tries to perform the transfer anyway:

```
// ./index.js
const { sequelize , BankAccount } = require("./models");

// This code will try to transfer $7,500 from Markov to Curie.
async function main() {
  // Fetch Markov and Curie's accounts.
  const markovAccount = await BankAccount.findByPk(1);
  const curieAccount = await BankAccount.findByPk(2);

  try {
    // Increment Curie's balance by $7,500.
    curieAccount.balance += 7500;
    await curieAccount.save();

    // Decrement Markov's balance by $7,500.
    markovAccount.balance -= 7500;
    await markovAccount.save();
  } catch (err) {
    // Report if anything goes wrong.
    console.log("Error!");

    for (const e of err.errors) {
      console.log(
        `${e.instance.clientName}: ${e.message}`
      );
    }
  }

  await sequelize.close();
}

main();
```

Running this code prints the following:

```
Executing (default): SELECT "id", "clientName", "balance", "createdAt", "updatedAt" FI
Executing (default): SELECT "id", "clientName", "balance", "createdAt", "updatedAt" FI
Executing (default): UPDATE "BankAccounts" SET "balance"=$1,"updatedAt"=$2 WHERE "id"
Error!
Markov: balance must not be less than zero
```

Everything starts out fine. We fetch Markov and Curie's accounts. We increase Curie's balance. But then we hit a snag: when we call `markovAccount.save()`, Sequelize detects that we are trying to set Markov's balance below zero. Sequelize therefore **does not** send a SQL request to update Markov's account balance. Instead, `markovAccount.save()` throws an exception. We print the error: Markov's balance must not be less than zero.

We thus avoid saving a negative balance for Markov. But other damage has already been done. If we now check account balances, we will see:

```
catsdb=> SELECT * FROM "BankAccounts";
 id | clientName | balance | ...
----+------------+---------+-----
  1 | Markov     |    5000 | ...
  2 | Curie      |   17500 | ...
(2 rows)
```

The bank has given free money to Curie! We should have "rolledback" the increase of Curie's balance. We will learn how to do that!

Incorrect Solutions

One may suggest a fix: make sure to decrement Markov's account balance before incrementing Curie's balance! If Markov's balance is insufficient, we can stop the transfer before giving Curie any money.

We *could* swap the order of the updates, and it would indeed fix this specific problem. But imagine if Markov tries to transfer $2,500 (an amount he can afford). We first decrement Markov's account balance and then -- the operating system crashes before the second update can be submitted. Curie's balance is not incremented. The bank has stolen Markov's money!

The problem is fundamental: no matter what order we perform the two updates in, the database can always fail *after* processing the first, but *before* processing the second. For our code to be resilient to unavoidable failures, there is no other choice but to use a database transaction.

Using A Database Transaction With Sequelize

Let's return to our previous example of trying to transfer $7,500 from Markov to Curie. Specifically, we will rewrite this key part:

```
// Increment Curie's balance by $7,500.
curieAccount.balance += 7500;
await curieAccount.save();

// Decrement Markov's balance by $7,500.
markovAccount.balance -= 7500;
await markovAccount.save();
```

To ask Sequelize to perform the two updates in a SQL database transaction, we use the `sequelize.transaction` method. We will write this like so, instead:

```
await sequelize.transaction(async (tx) => {
  // Increment Curie's balance by $7,500.
  curieAccount.balance += 7500;
  await curieAccount.save({ transaction: tx });

  // Decrement Markov's balance by $7,500.
  markovAccount.balance -= 7500;
  await markovAccount.save({ transaction: tx });
});
```

Let's go through the transaction code and explain each part:

```
// Start a transaction. Queries run inside the callback can be part of
// the transaction.
await sequelize.transaction(async (tx) => {
  // Increment Curie's balance by $7,500.
  curieAccount.balance += 7500;
  // Pass the `tx` transaction object so that Sequelize knows to
  // update Curie's account as part of this transaction (rather than
  // "on its own" per usual).
  await curieAccount.save({ transaction: tx });

  // Decrement Markov's balance by $7,500.
  markovAccount.balance -= 7500;
  // Again, pass the `tx` transaction object. Thus both updates are part
  // of the same transaction.
  await markovAccount.save({ transaction: tx });

  // If no exceptions have been thrown, `sequelize.transaction` will
  // `COMMIT` the transaction after the end of the callback.
  //
  // If any error gets thrown, `sequelize.transaction` will abort
  // the transaction by issuing a `ROLLBACK`. This will cancel all
  // updates.
});
```

Let's put the transaction code back into our original program:

```
// ./index.js
const { sequelize, BankAccount } = require("./models");

async function main() {
  // Fetch Markov and Curie's accounts.
  const markovAccount = await BankAccount.findByPk(1);
  const curieAccount = await BankAccount.findByPk(2);

  try {
    await sequelize.transaction(async (tx) => {
      // Increment Curie's balance by $7,500.
      curieAccount.balance += 7500;
      await curieAccount.save({ transaction: tx });

      // Decrement Markov's balance by $7,500.
      markovAccount.balance -= 7500;
      await markovAccount.save({ transaction: tx });
    });
  } catch (err) {
    // Report if anything goes wrong.
    console.log("Error!");

    for (const e of err.errors) {
      console.log(
        `${e.instance.clientName}: ${e.message}`
      );
    }
  }

  await sequelize.close();
}

main();
```

Running this code prints:

```
Executing (default): SELECT "id", "clientName", "balance", "createdAt", "updatedAt" FI
Executing (default): SELECT "id", "clientName", "balance", "createdAt", "updatedAt" FI
Executing (208b3951-9ab9-489b-97f0-afb49aaff807): START TRANSACTION;
Executing (208b3951-9ab9-489b-97f0-afb49aaff807): UPDATE "BankAccounts" SET "balance"=
Executing (208b3951-9ab9-489b-97f0-afb49aaff807): ROLLBACK;
Error!
Markov: balance must not be less than zero
```

Let's review what happened. We again start by fetching both `BankAccount`s. We next `START TRANSACTION`. We issue the update to Curie's account.

Then Sequelize detects the validation failure when trying to run `markovAccount.save({ transaction: tx })`. Markov doesn't have enough money in his account! Sequelize throws an exception. The `sequelize.transaction` method *catches the exception* and issues a `ROLLBACK` for the transaction. This tells the database to undo the prior increment of Curie's account balance.

Having rolled back the transaction, the `sequelize.transaction` method *rethrows* the error, so that our logging code gets a chance to learn about the error and print its details.

Aside: What Is The `Transaction` Object?

This is bonus information in case you are troubled by what the `tx` parameter to `sequelize.transaction` is for. You can use transactions correctly without knowing this bonus information.

What is the mysterious `tx` that is passed by `sequelize.transaction` to our callback? It is basically just a unique ID. In this case, the ID is: `208b3951-9ab9-489b-97f0-afb49aaff807`. You can see this ID in the logs above.

When we say `curieAccount.save({ transaction: tx })`, we are telling Sequelize: "update Curie's account as part of transaction number `208b3951-9ab9-489b-97f0-afb49aaff807`."

Sequelize needs transaction IDs because it can be running many SQL transactions *concurrently* (loosely speaking: "in parallel"). One part of the application could be transferring money from Markov to Curie at the same time another part of the application is transferring money from Kate to Ned.

If Sequelize did not keep track of transaction IDs, it would not know that `curieAccount.save()` should be a part of the Markov/Curie transaction rather than the Kate/Ned transaction.

Transactions Prevent *Race Conditions*

There is still a subtle mistake in my bank transfer code. There is a potential problem if someone modifies Markov's or Curie's account in between (1) the initial fetch of their accounts, and (2) the transaction to update the accounts.

```
// ./index.js
async function main() {
  // I will transfer only $5,000 so that Markov's balance can cover the
  // amount. Markov starts out with $5,000.

  // Fetch Markov and Curie's accounts.
  const markovAccount = await BankAccount.findByPk(1);
  const curieAccount = await BankAccount.findByPk(2);

  // ***
  // Imagine that right now some other program transfers the $5,000 out
  // of Markov's account. Markov's true account **in the database** now
  // has a balance of $0. But `markovAccount.balance` is still $5,000,
  // because we fetched Markov's `BankAccount` **before** the transfer
  // was made!
  // ***

  try {
    await sequelize.transaction(async (tx) => {
      // Increment Curie's balance by $5,000 (to $15,000).
      curieAccount.balance += 5000;
      await curieAccount.save({ transaction: tx });

      // Decrement `markovAccount.balance` by $5,000.
      // `markovAccount.balance` is set to zero.
      markovAccount.balance -= 5000;
      // Save and set Markov's balance to zero.
      await markovAccount.save({ transaction: tx });

      // Problem: Markov's balance in the database was *already* zero.
      // Markov had no money to transfer. He should not have been able
      // to transfer the $5,000.
    });
  } catch (err) {
    // ...
  }

  await sequelize.close();
}

main();
```

Because another program can "race in between" (1) the reading of the account balances and (2) the updating of the balances, we call this potential problem a *race condition*. The easiest way to fix the race condition is to prohibit anyone else from modifying Markov's account balance in between (1) and (2).

Luckily, the solution is simple. Any data used in a transaction will be *locked* until the transaction completes. Data that is locked can be neither read nor written by other transactions. If our transaction reads (or writes) data, no one else can read or write that data until our transaction completes. When we `COMMIT` (or `ROLLBACK`) all the locked data is freed (the locks are *released*).

We don't have to lock the data ourselves. Simply by doing all our queries inside the same transaction, the database will lock the data for us. Therefore, to fix the problem, we should move the initial account fetching by `findByPk` into the transaction (i.e., pass it `{ transaction: tx }`):

```
async function main() {
  try {
    // Do all database access within the transaction.
    await sequelize.transaction(async (tx) => {
      // Fetch Markov and Curie's accounts.
      const markovAccount = await BankAccount.findByPk(
        1, { transaction: tx },
      );
      const curieAccount = await BankAccount.findByPk(
        2, { transaction: tx }
      );

      // No one can mess with Markov or Curie's accounts until the
      // transaction completes! The account data has been locked!

      // Increment Curie's balance by $5,000.
      curieAccount.balance += 5000;
      await curieAccount.save({ transaction: tx });

      // Decrement Markov's balance by $5,000.
      markovAccount.balance -= 5000;
      await markovAccount.save({ transaction: tx });
    });
  } catch (err) {
    // ...
  }

  await sequelize.close();
}

main();
```

This prints:

```
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): START TRANSACTION;
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): SELECT "id", "clientName", "balance"
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): SELECT "id", "clientName", "balance"
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): UPDATE "BankAccounts" SET "balance"=
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): UPDATE "BankAccounts" SET "balance"=
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): COMMIT;
```

Notice that now *everything* is done in the transaction `76321a03-93c5-47c0-861a-cf24c3e6f3bf`. This includes the initial fetching of the accounts. Because the fetching is done within the transaction, other users are not allowed to modify the accounts until the transaction is finished.

Moving our read operations into the transaction has solved our race condition problem. Every Sequelize operation - whether reading or writing - can take a `transaction: tx` option. This includes:

1. `findByPk`
2. `findAll`
3. `save`
4. `create`
5. `destroy`

Conclusion

Having completed this reading, here are the important things to take away:

1. You should know that any SQL update operation may fail, often through no fault of your own.

2. You should know that you must write code resilient to SQL failures.
3. You should know that when performing multiple update operations as part of a group, you must use a transaction.
4. You should know how to use `sequelize.transaction` to run commands within a SQL transaction.
5. You should know how to pass a transaction object as the `{ transaction: tx }` parameter to a Sequelize command (such as `save`).
6. You should know what a *race condition* is: the possibility that someone else modifies previously fetched data before you are finished using/updating it.
7. You should know how to use transactions to guard against race conditions. That is: you should know that both reading and writing operations should be put in the same transaction.

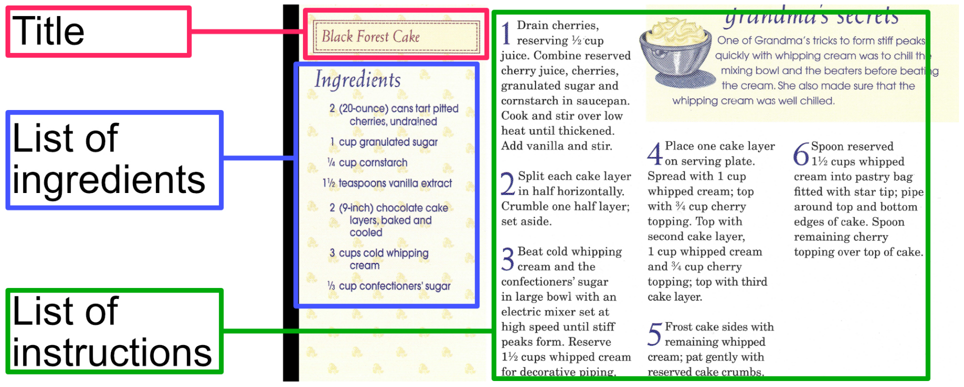
Recipe Box Project

In this project, you will build the Data Access Layer to power a Web application. This means that you will provide all of the SQL that it takes to get data into and from the database. You will do this in SQL files that the application will load and read.

Note: This is not a good way to create a database-backed application. This project is structured this way so that it isolates the activity of SQL writing. Do not use this project as a template for your future projects.

The data model analysis

What goes into a recipe box? Why, recipes, of course! Here's an example recipe
card.


Title
List of ingredients
List of instructions

Black Forest Cake

Ingredients

2 (20-ounce) cans tart pitted cherries, undrained

1 cup granulated sugar

¼ cup cornstarch

1½ teaspoons vanilla extract

2 (9-inch) chocolate cake layers, baked and cooled

3 cups cold whipping cream

⅓ cup confectioners' sugar

1 Drain cherries, reserving ½ cup juice. Combine reserved cherry juice, cherries, granulated sugar and cornstarch in saucepan. Cook and stir over low heat until thickened. Add vanilla and stir.

2 Split each cake layer in half horizontally. Crumble one half layer; set aside.

3 Beat cold whipping cream and the confectioners' sugar in large bowl with an electric mixer set at high speed until stiff peaks form. Reserve 1½ cups whipped cream for decorative piping.

4 Place one cake layer on serving plate. Spread with 1 cup whipped cream; top with ¾ cup cherry topping. Top with second cake layer, 1 cup whipped cream and ¾ cup cherry topping; top with third cake layer.

5 Frost cake sides with remaining whipped cream; pat gently with reserved cake crumbs.

6 Spoon reserved 1½ cups whipped cream into pastry bag fitted with star tip; pipe around top and bottom edges of cake. Spoon remaining cherry topping over top of cake.

Grandma's secrets

One of Grandma's tricks to form stiff peaks quickly with whipping cream was to chill the mixing bowl and the beaters before beating the cream. She also made sure that the whipping cream was well chilled.

You can see that a recipe is made up of three basic parts:

- A title,
- A list of ingredients, and
- A list of instructions.

You're going to add a little more to that, too. It will also have

- The date/time that it was entered into the recipe box, and
- The date/time it was last updated in the recipe box.

These are good pieces of data to have so that you can show them "most recent"
for example.

Ingredients themselves are complex data types and need their own structure. They
"belong" to a recipe. That means they'll need to reference that recipe. That means an ingredient is made up of:

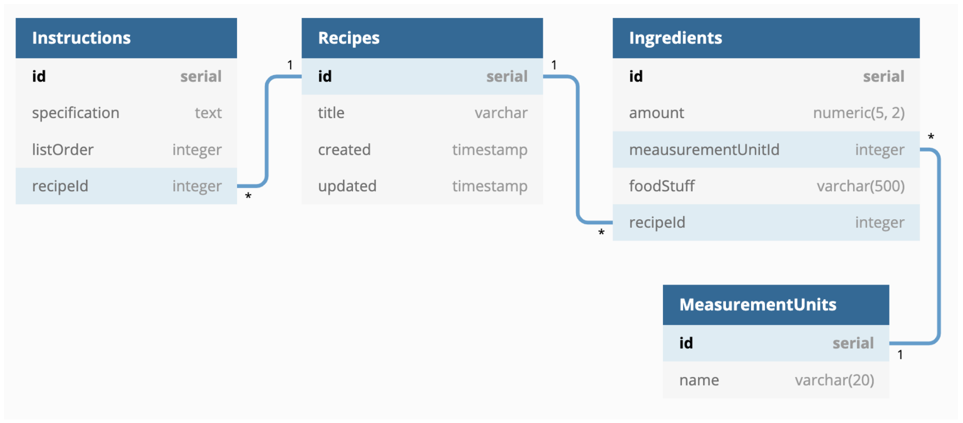
- An amount (optional),
- A unit of measure (optional),
- The actual food stuff, and
- The id of the recipe that it belongs to.

That unit of measure is a good candidate for normalization, don't you think? It's a predefined list of options that should not change and that you don't want people just typing whatever they want in there, not if you want to maintain data integrity. Otherwise, you'll end up with "C", "c", "cup", "CUP", "Cup", and all the other permutations, each of which is a distinct value but means the same
thing.

Instructions are also complex objects, but not by looking at them. Initially, one might only see text that comprises an instruction. But, very importantly, instructions have *order*. They also *belong* to the recipe. With that in mind, an instruction is made up of:

- The text of the instruction,
- The order that it appears in the recipe, and
- The id of the recipe that it belongs to.

That is enough to make a good model for the recipe box.



Getting started

- Download the starter project from https://github.com/appacademy-starters/sql-recipe-box
- Run `npm install` to install the packages
- Run `npm start` to start the server on port 3000

You'll do all of your work in the **data-access-layer** directory. In there, you will find a series of SQL files. In each, you will find instructions of what to do to make the user interface to work. They are numbered in an implied order for
you to complete them. The only real requirement is that you finish the SQL for the **00a-schema.sql** and **00b-seed.sql** files first. That way, as you make your way through the rest of the SQL files, the tables and some of the data will
already exist for you. You can run the command `npm run seed` to run both of those files or pipe each it into `psql` as you've been doing.

Either way that you decide to seed the database, you'll need to stop your server. The seed won't correctly run while the application maintains a connection to the application database. You may also need to exit all of the active `psql` instances that you have running to close out all of the active connections. When you run the seed, if it reports that it can't do something because of active connections, look for a running instance of the server, Postbird, or `psql` with that connection.

Warning: running the seed files will destroy all data that you have in the database.

Your SQL

When you write the SQL, they will mostly be parameterized queries. The first couple of files will show you how it needs to be done, where you will place the parameter placeholders "1”,”2", and so on. If you need to, refer to the Parameterized query section of the documentation for **node-postgres**.

In each of the following files in the **data-access-layer**, you will find one or more lines with the content `-- YOUR CODE HERE`. It is your job to write the SQL statement described in the block above that code. Each file is named with
the intent of the SQL that it should contain.

The application

The application is a standard express.js application using the pug library to generate the HTML and the node-postgres library to connect to the database.

It reads your SQL files whenever it wants to make a database call. If your file throws an error, then the UI handles it by telling you what needs to be fixed so that the UI will work. The application will also output error messages for missing functionality.

The SQL files contain a description of what the content is and where it's used in the application. Tying those together, you'll know you're done when you have
all of the SQL files containing queries and there are no errors in the UI or console.

Directions

Fill out the **00a-schema.sql** and **00b-seed.sql** files first. Then seed the database with command, `npm run seed`.

Home Page

Start the server by running `npm run dev`. Then go to `localhost:3000` you should
see the home page with "Latest Recipes". To show the latest recipes properly, complete the `01-get-ten-most-recent-recipes.sql` file.

After completing the file, make sure you correctly defined the sql query so that the first recipe listed is the most recently updated recipe.

`/recipes/:id`

If you click on one of the recipes in the list of recipes on the home page, it will direct you to that recipe's Detail Page. Complete the following files that correspond to this page and make sure to test a file right after you fill out the file by refreshing the page:

- `02a-get-recipe-by-id.sql`
- `02b-get-ingredients-by-recipe-id.sql`
- `02c-get-instructions-by-recipe-id.sql`

Make sure to read the instructions well! In all the above sql queries, the `$1` parameter will be the recipe id.

`/recipes/new`

Click on `ADD A RECIPE` button on the Navigation Bar to direct you to the New Recipe Form page. Fill out the `03a-insert-new-recipe.sql` file so you can
create a new recipe.

`/recipes/:id/edit`

After creating a new recipe, you will be directed to the Recipe Edit page where
you can add instructions and ingredients to a recipe. Complete the following files that correspond to this page and make sure to test a file right after
you fill out the file:

- `03b-get-units-of-measure.sql`
- `04-insert-new-ingredient.sql`
- `05-insert-new-instruction.sql`
- `06-delete-recipe.sql`

`/recipes?term={searchTerm}`

Allow users to find recipes by a part of their name using the Search Bar in the Navigation Bar. Complete `07-search-recipes.sql` for this feature.