WEEK 10
Relational Databases

Make It Pretty Learning Objectives

- **Make It Pretty Practice**
  - What Recruiters Are Looking For
  - Attributes of Great Looking Websites
  - Additional requirements
  - Avoid these things
  - Exercise

RDBMS And Database Entity Learning Objectives

- **Database Lingo: Relational Database Management System**
  - RDBMS
  - What is PostgreSQL?
  - What is SQL?
- **Installing PostgreSQL 12 and Postbird on Windows**
- **Installing PostgreSQL 12 and Postbird on macOS**
- **User Management Walk-Through**
  - Naming a user
  - Creating a superuser
  - Creating a limited user
  - Removing a user
  - Case sensitivity
- **Database Management Walk-Through**
  - Naming a database
  - Creating a database for your user
  - Creating other users and databases
  - Applying security to databases
  - Listing databases and granting privileges
  - Time to clean up
- **Table Management Walk-Through - Part I**
  - What is a table?
  - Defining tables
  - Other data types
  - Naming a table
  - Writing the SQL
  - Listing tables and table definitions
  - Deleting a table
- **Table Management Walk-Through - Part II**
  - Nullable columns
  - Default values
  - Primary keys
  - Unique values
  - Refactor for data integrity
  - Order of table declarations
- *Database Management Project*

SQL Learning Objectives

- **Retrieving Rows From A Table Using SELECT**
  - What is a query?
  - Example table
  - Using psql in the terminal
  - Simple SELECT Query
  - Formatting SELECT statements
- **Selecting Table Rows Using WHERE And Common Operators**
  - Using SELECT and WHERE
- **Inserting Data Into A Table**

- **Foreign Keys And The JOIN Operation**
  - Setting up the database
  - Using JOIN to retrieve rows from multiple tables
  - Helpful links:
- **Writing And Running A Seed File In PSQL**
  - Creating a seed file
  - Populating a database via < (“left caret”)
  - Populating the database via | (“pipe”)
- *Create And Seed A Database Project*
- *Solving The SQL Menagerie Project*
- **Creating A Schema For Relational Database Design**
  - What is Relational Database Design?
  - Stages of Relational Database Design
  - Schema design tools
- **Using SQL Transactions**
  - What is a transaction?
  - Implicit vs. explicit transactions
  - When to use transactions and why
  - Transaction properties: ACID
  - Helpful links:
- **Joins vs. Subqueries**
  - What is a JOIN?
  - What is a subquery?
  - Using joins vs. subqueries
  - Helpful links:
- **PostgreSQL Indexes**
  - What is a PostgreSQL index?
  - How to create an index
  - When to use an index
- **Node-Postgres And Prepared Statements**
  - Connecting
  - Prepared Statement
  - Try it out for yourself
- *Recipe Box Project*

ORM Learning Objectives

- **Installing And Using Sequelize**
  - What Is An ORM?
  - How To Install Sequelize
  - How To Initialize Sequelize
  - Verifying That Sequelize Can Connect To The Database
  - Our Preexisting Database Schema
  - Using Sequelize To Generate The Model File
  - Examining (And Modifying) A Sequelize Model File
  - Using The `Cat` Model To Fetch And Update SQL Data
  - Reading And Changing Record Attributes
- **Using Database Migrations**
  - Sequelize Migration Files
  - Running A Migration
  - Rolling Back A Migration
  - Editing A Migration File
  - `up` And `down` are Asynchronous
  - Writing A `down` Method
  - Advantages Of Migrations
- **CRUD Operations Using Sequelize**
  - Creating A New Record
  - Reading A Record By Primary Key
  - Updating A Record
  - Destroying A Record
  - Class Methods For CRUD
- **Querying Using Sequelize**
  - Basic Usage Of `findAll` To Retrieve Multiple Records
  - Using `findAll` To Find Objects Not Matching A Criterion
  - Combining Criteria with `Op.and`

WEEK-10 DAY-1
Hello, Database!

Make It Pretty Learning Objectives

It is really important that you make the best impression that you can with the projects that you will soon start. To that end, the objectives for your learning with this section should allow you to

- Recall the items recruiters are most interested
- Explain the aspects of good looking Web application
- Identify App Academy's expectations of your projects for after you graduate
- Practice good code hygiene when making projects live

Make It Pretty Practice

Your portfolio projects are the first impression that a company has of you. Imagine you are a non-technical person looking to hire a developer to build your website. What would you think if you reviewed a site that looked unfinished, or poorly styled, even if the functionality worked perfectly? Would you have the perspective to tell the difference from a poorly implemented site and one that is robust, but not visually polished?

Non-technical people will be looking at your projects before engineers, and they can't see the amount of work it takes to get a backend up and running. All they see is the visual appearance, so unless you take care of your frontend you'll never even get the chance to talk about the backend work you did.

This material should make it so that you can

- Evaluate your site against industry-standard visual styles
- Identify the attributes of current trends in website presentation
- Identify gaps that can cause a website to be perceived as incomplete

What Recruiters Are Looking For

Recruiters expect professionalism and good design (we'll discuss more about what good design means below). A good litmus test is: if you were to stumble upon your portfolio sight unexpectedly, would you be able to tell that this wasn't done by a professional dev? In other words, does your website look on par with the millions of other production sites that exist on the internet?

In response to what recruiters and interviewers might ask about how you choose different styles for your Web applications, review TopTal's 12 Essential Web Interview Questions.

Attributes of Great Looking Websites

Unless you know exactly what you are doing when deciding on a visual approach, you should select and follow a modern design framework, or use a template.

The first thing to pay attention to is padding and margin. Every element should have padding so that its inner contents are not butting up against its edges and margin so that the element itself is not butting up against any other elements. In general sibling elements should NOT touch or overlap. A good way to estimate the correct amount is to imagine a lower-case `a` in the same size and font of nearby text. You should be able to fit this `a` in both the padding and the margin such that it just barely touches the edge/text on each side.

Be sure to balance whitespace when laying out your elements and adding margin/padding. If you have 20px of whitespace to the left of the element you should probably have 20px to the right as well. Make sure things are centered correctly (horizontally and vertically) and be consistent! I.E. If you have a row of buttons in the header they should all be aligned vertically with consistent margin and padding.

Use a color palette to determine your website's themes and avoid color clashes. You should have a primary color, a secondary color, and 1 or 2 accent colors. Your primary color is going to be the most abundant on the site followed by your secondary color. The accent color is used for things like buttons, tools, and other areas that you want to draw the user's eye to. Use it sparingly!

Use Google Font Pairing recommendations to find good fonts. In general, you should not have more than 2 fonts in a web app and you should avoid mixing serif and sans-serif fonts.

Pay attention to font-size and weight! You should use textual hierarchies to break up your text and make it easier to read. Prefer multiple short lines to fewer long lines of text when displaying info to the user. Your headers should be large and paragraphs should be slightly smaller font size. Having widely varying and inconsistent font sizes is one of the surest signs that a website was designed by a beginner. When in doubt, simplify.

Make sure your color and text choices pass contrast requirements.

Most modern websites slightly round the corners of buttons and background cards/modals. You should, too. Also, take advantage of advanced CSS features like transitions and shadows to make your site pop. Make sure you let your user know what parts of the site are alive through affordances.

Additional requirements

In addition to the above recommendations, App Academy also expects your projects to include the following:

- **Seed Data**: Make sure your seeds are plentiful and appropriate. Even an excellently designed site will fail to impress if you don't have a good seed data.
- **Favicon**: Make sure your website has a favicon.
- **Demo Login**: Make sure your website has a demo login. Most recruiters will not sign up for your website with their own email address as the chance for

misuse of said email address is too high.
- **Console output**: Be sure that your console is free of logs and error messages. Nothing screams amateur more than seeing a ton of console.logs when
visiting a potential candidate's website.
- **Personal Links**: Any links to your GitHub, LinkedIn, or Angel List should open in a new tab.
- **Scorecard**: After you graduate, you'll stop using Progress Tracker and start using InterviewDB, another one of our applications. When you go to turn in your project on InterviewDB, be sure to include the scorecard so that your advisor can grade your work. Note that you'll need to make your own copy and save it to your google drive before adding the link to InterviewDB.

Avoid these things

We have collected a lot of feedback from recruiters and hiring managers over the years. These are the tips and tricks that they tell us will turn them off to reviewing a student's project.

- Avoid fonts that look like handwriting
- Avoid over using accent colors
- Avoid themes that relate to specific holidays (ie. don't make a Christmas themed app)
- Dead Links. If you didn't implement a feature then don't put a link to it. If you do, then you're forcing recruiters to find the needle of implemented features in a haystack of unimplemented features. Your site will be perceived as incomplete or broken
- Avoid linking to the actual site that you are cloning
- Avoid neon, bright, or crazy colors
- Avoid having too many different colors in your app.
- Avoid putting affordances on things that can't be clicked or interacted with
- Avoid blinking, spinning or flashing images
- Avoid busy tiled background images with any color text
- Avoid having everything centered. When in doubt, do not center. Do not center more than three lines of text
- Avoid too many images or huge images. Minify all images that are not of a product and do not need to be examined closely
- Avoid long lists of links
- Avoid too many headlines
- Do not use blinking or flashing text, images, or transitions

Exercise

Select three sites from Product Hunt. For each site, list the:

- Fonts Used
- Colors in Palette
- Contrast ratio of the main text and it's background (Use the contrast checker)
- The size of padding/margins compared to a lowercase 'a'
- The maximum number of lines in a row center-justified anywhere on the site
- Load Time
- Theme or style based on (Bootstrap, Material, etc.)
- Number of broken or under construction pages

RDBMS And Database Entity Learning Objectives

Databases are an essential part of many Web applications. There are lots of things we could store in a database and use in a Web app, including user information, product information, review information, and more. Learning how to

create databases and retrieve information stored in a database to display in a Web app is a foundational development skill.

In this section, you will be able to:

- Define what a relational database management system is
- Describe what relational data is
- Define what a database is
- Define what a database table is
- Describe the purpose of a primary key
- Describe the purpose of a foreign key
- Describe how to properly name things in PostgreSQL
- Install and configure PostgreSQL 12, its client tools, and a GUI client for it named Postbird
- Connect to an instance of PostgreSQL with the command line tool `psql`
- Identify whether a user is a normal user or a superuser by the prompt in the `psql` shell
- Create a user for the relational database management system
- Create a database in the database management system
- Configure a database so that only the owner (and superusers) can connect to it
- View a list of databases in an installation of PostgreSQL
- Create tables in a database
- View a list of tables in a database
- Identify and describe the common data types used in PostgreSQL
- Describe the purpose of the UNIQUE and NOT NULL constraints, and create columns in database tables that have them
- Create a primary key for a table
- Create foreign key constraints to relate tables
- Explain that SQL is not case sensitive for its keywords but is for its entity names

---

Database Lingo
The Relational Database Management System

Databases are an essential part of many Web applications. There are lots of things we could store in a database and use in a Web app, including user information, product information, review information, and more. Learning how to create databases and retrieve information stored in a database to display in a Web app is a foundational development skill.

The most popular kind of database is called a *relational database*. That's what you'll primarily use in this course. There are other databases called "document databases", "non-relational databases", or "NoSQL databases" that have become popular since the mid-2000s. They serve a different purpose that relational databases by storing data in ways that are different than the way relational databases store their data.

In this reading, you will learn about **relational database management systems**. Then, you will install one. Then, you will start using it!

RDBMS

That's quite an ugly acronym, but it's what developers have when referring to the software application that manages databases for us. Here's an important difference for you to understand.

The **RDBMS** is a software application that you run that your programs can connect to that they can store, modify, and retrieve data. The RDBS that you will use in this course is called PostgreSQL, often shortened to just "postgres", pronounced like it's spelled. It is an "open-source" RDBMS which

means that you can go read the source code for it, copy it, modify it, and make your own specialized version of an RDBMS. Often, developers will talk about the "database server". That is the computer on which the RDBMS is running.

A **database** (or more properly **relational database**) is a collection of structured data that the RDBMS manages for you. A single running RDBMS can have hundreds of databases in it that it manages.

Software developers will often use the term "database" to refer to the RDBMS. They will also say that "the data is in postgres" or "the data is in Oracle" which is terribly ambiguous because those are the names of the RDBMSes, not a database where the data is stored. That'd be like asking someone their address and them replying "Chicago".

Just be aware that the language around these terms is loose.

What is PostgreSQL?

Again, PostgreSQL is software. Specifically, it is an open-source, relational database management system. It is derived from the POSTGRES package written at UC Berkeley. The specific name “PostgreSQL” was coined in 1996, after SQL was implemented as its core query language. PostgreSQL provided a new program (new for 1996) for interactive SQL queries called `[psql]`, which is terminal-based front-end to PostgreSQL that lets you to type in queries interactively, issue them to PostgreSQL, and see the query results.

You install PostgreSQL onto a computer. It then runs as a "service", that is, a program that runs in the background that should not stop until the machine does. You will install it on your computer. It will quietly run in the background, eagerly awaiting for you to connect to it and interact with it from the command line, from a GUI tool, or from your application.

When you do connect with it, you will interact with it through a small set of its own commands and SQL.

What is SQL?

SQL (pronounced "sequel" or "s-q-l") stands for "Structured Query Language". It is not a programming language like JavaScript. JavaScript, as you well know, has *control flow*, with `for` loops and `if` statements. Most SQL that you write doesn't have all that. Instead, it is a *declarative* programming language. You tell the database what computation you want it to do, and it does it. In that way, SQL is more like CSS than JavaScript.

Whereas JavaScript works on variables and arrays of single values, most SQL works on *sets* of records. You'll see more what that means later, but just know that in the SQL that you learn, you won't declare a single variable in that SQL.

SQL, the language, is the primary way that you will interact with the RDBMS to affect the data in a single database or the structure of the database itself. The process of using SQL takes two steps:

1. Connect to an RDBMS specifying
    - credentials, user name and password
    - the name of the database that you want to use
2. Issue one or more SQL statements to interact with
    - the structure of the database
    - the data in the database

Some vendor-specific variants of SQL *do* have loops and if-statements. However, you will be learning the general kind of SQL, the one managed by the American National Standards Institute, called ANSI SQL which defines the way that we get data out of, put data into, modify data in, and remove data from a database. This type of SQL is *portable* between different types of database management systems. That means most of what you learn in this course, you will be able to

use on *any* relational database management system that supports ANSI SQL, RDBMSes such as

- **Informix**, a commercial RDBMS from IBM intended to run on servers and mainframes
- **Microsoft Access**, a commercial RDBMS from Microsoft intended for personal use
- **Microsoft SQL Server**, a commercial RDBMS from Microsoft intended to run on servers
- **MySQL**, an open-source RDBMS comparable to PostgreSQL
- **Oracle DB**, a commercial RDBMS from Microsoft intended to run on servers
- **SQLite**, an open-source RDBMS that many applications *embed* into the program to efficiently store the application's data relational data

Now that this preamble is out of the way, the next step is to install PostgreSQL!

What we learned:

In this article you learned that

- A *relational database management* system is software that usually runs as a service that manages collections of structured data called databases.
- PostgreSQL is one of many relational database management systems and the one
  that you will mostly use in this course
- A *database* is a collection of structured data that an RDBMS manages for you.
- The Structured Query Language (SQL) is a programming language that allows you
  to interact with the structure of the database and the actual data that's stored in it.

Installing PostgreSQL 12 and Postbird on Windows

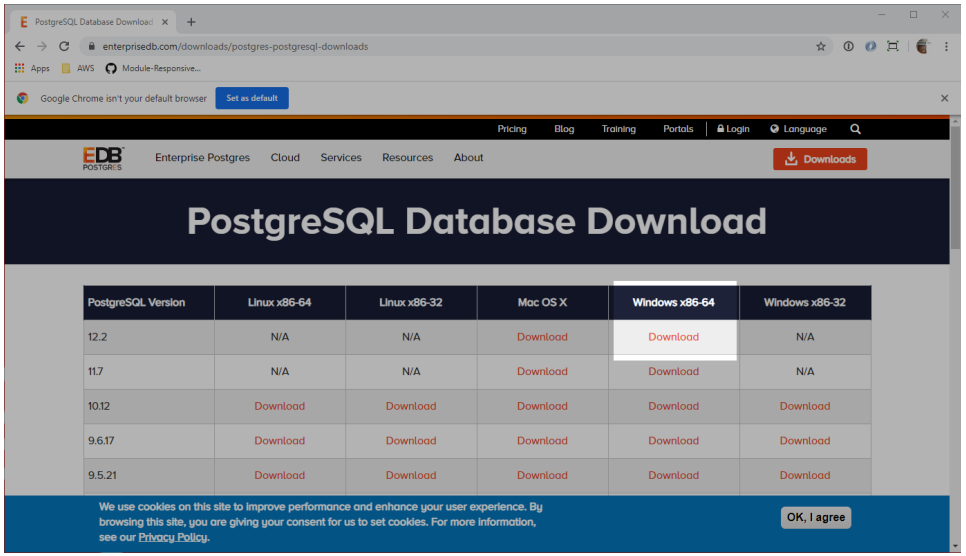
You will install three pieces of software so that you can start using PostgreSQL. You will install PostgreSQL itself on your Windows installation. Then, you will install `psql` in your Ubuntu installation. Then you will also install Postbird, a cross-platform graphical user interface that makes working with SQL and PostgreSQL better than just using the command line tool `psql`.

When you read "installation", that means the actual OS that's running on your machine. So, you have a Windows installation, Windows 10, that's running when you boot your computer. Then, when you start the Ubuntu installation, it's as if there's a completely separate computer running inside your computer. It's like having two completely different laptops.

Installing PostgreSQL 12

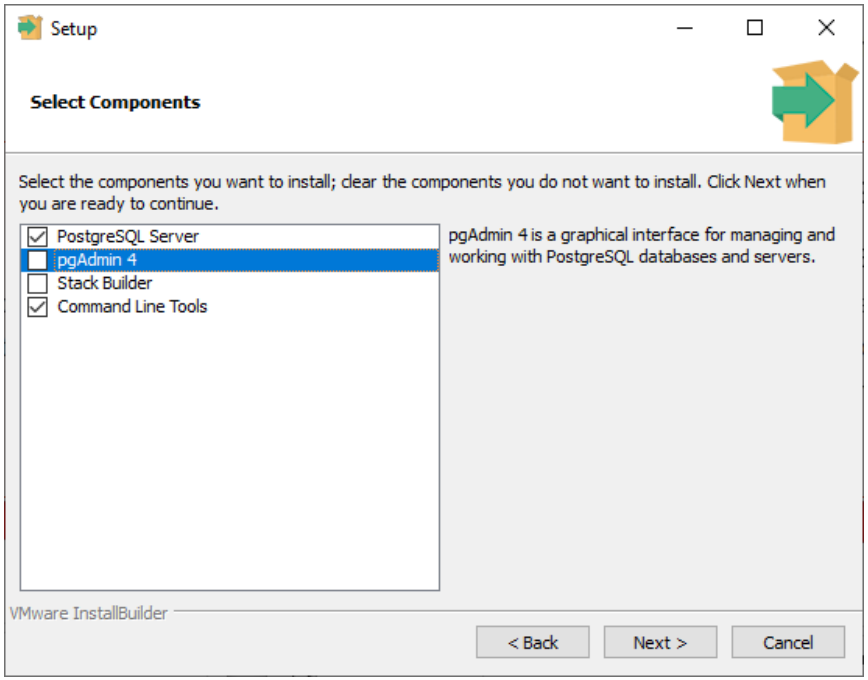
To install PostgreSQL 12, you need to download the installer from the Internet. PostgreSQL's commercial company, Enterprise DB, offers installers for PostgreSQL for every major platform.

Open https://www.enterprisedb.com/downloads/postgres-postgresql-downloads. Click the link for PostgreSQL 12 for Windows x86-64.



Once that installer downloads, run it. You need to go through the normal steps of installing software.

- Yes, Windows, let the installer make changes to *my* device.
- Thanks for the welcome. Next.
- Yeah, that's a good place to install it. Next.
- I don't want that pgAdmin nor the Stack Builder things. Uncheck. Uncheck. Next.



- Also, great looking directory. Thanks. Next.
- Oooh! A password! I'll enter ********. I sure won't forget that because, if I do, I'll have to uninstall and reinstall PostgreSQL and lose all of my hard work. **Seriously, write down this password or use one you will not forget.** Next.
- Sure. 5432. Good to go. Next.
- Not even sure what that means. Default! Next.
- Yep. Looks good. Next.
- Geez. Really? Thanks. Next.
- *Time to get a tea.*
- All right! All done. Finish!

Installing PostgreSQL Client Tools on Ubuntu

Now, to install the PostgreSQL Client tools for Ubuntu. You need to do this so that the Node.js (and later Python) programs running on your Ubuntu installation can access the PostgreSQL server running on your Windows installation. You need to tell `apt`, the package manager, that you want it to go find the PostgreSQL 12 client tools from PostgreSQL itself rather than the common package repositories. You do that by issuing the following two commands. Copy and paste them one at a time into your shell. (If your Ubuntu shell isn't running, start one.)

Pro-tip: Copy those commands because you're not going to type them, right? After you copy one of them, you can just right-click on the Ubuntu shell. That should paste them in there for you.

```
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-k
```

If prompted for your password, type it.

```
echo "deb http://apt.postgresql.org/pub/repos/apt/ `lsb_release -cs`-pgdg main" |
```

The last line of output of those two commands running should read "OK". If it does not, try copying and pasting them one at a time.

Now that you've registered the PostgreSQL repositories as a source to look for PostgreSQL, you need to update the `apt` registry. You should do this before you install *any* software on Ubuntu.

```
sudo apt update
```

Once that's finished running, the new entries for PostgreSQL 12 should be in the repository. Now, you can install them with the following command.

```
sudo apt install postgresql-client-12 postgresql-common
```

If it asks you if you want to install them, please tell it "Y".

Test that it installed by typing `psql --version`. You should see it print out information about the version of the installed tools. If it tells you that it can't find the command, try these instructions over.

Configuring the client tools

Since you're going to be accessing the PosgreSQL installation from your Ubuntu installation on your Windows installation, you're going to have to type that you want to access it over and over, which means extra typing. To prevent you from having to do this, you can customize your shell to always add the extra commands for you.

This assumes you're still using Bash. If you changed the shell that your Ubuntu installation uses, please follow that shell's directions for adding an alias to its startup file.

Make sure you're in your Ubuntu home directory. You can do that by typing `cd` and hitting enter. Use `ls` to find out if you have a `.bashrc` file. Type `ls .bashrc`. If it shows you that one exists, that's the one you will add the alias to. If it tells you that there is no file named that, then type `ls .profile`. If it shows you that one exists, that's the one you will add the alias to. If it shows you that it does not exist, then use the file name `.bashrc` in the following section.

Now that you know which profile file to use, type `code «profile file name»` where "profile file name" is the name of the file you determined from the last section. Once Visual Studio Code starts up with your file, at the end of it (or if you've already added aliases, in that section), type the following.

```
alias psql="psql -h localhost"
```

When you run `psql` from the command line, it will now always add the part about wanting to connect to *localhost* every time. You would have to type that each time, otherwise.

To make sure that you set that up correctly, type `psql -U postgres postgres`. This tells the `psql` client that you want to connect as the user "postgres" (`-U postgres`) to the database postgres (`postgres` at the end), which is the default database created when PostgreSQL is installed. It will prompt you for a password. Type the password that you used when you installed PostgrSQL, earlier. If the alias works correctly and you type the correct password, then you should see something like the following output.

```
psql (12.2 (Ubuntu 12.2-2.pgdg18.04+1))
Type "help" for help.

postgres=#
```

Type `\q` and hit Enter to exit the PostgreSQL client tool.

Now, you will add a user for your Ubuntu identity so that you don't have to specify it all the time. Then, you will create a file that PostgreSQL will use to automatically send your password every time.

Copy and paste the following into your Ubuntu shell. Think of a password that you want to use for your user. **Replace the password in the single quotes in the command with the password that you want.** It *has* to be a non-empty string. PostgreSQL doesn't like it when it's not.

```
psql -U postgres -c "CREATE USER `whoami` WITH PASSWORD 'password' SUPERUSER"
```

It should prompt you for a password. Type the password that you created when you installed PostgreSQL. Once you type the correct password, you should see "CREATE ROLE".

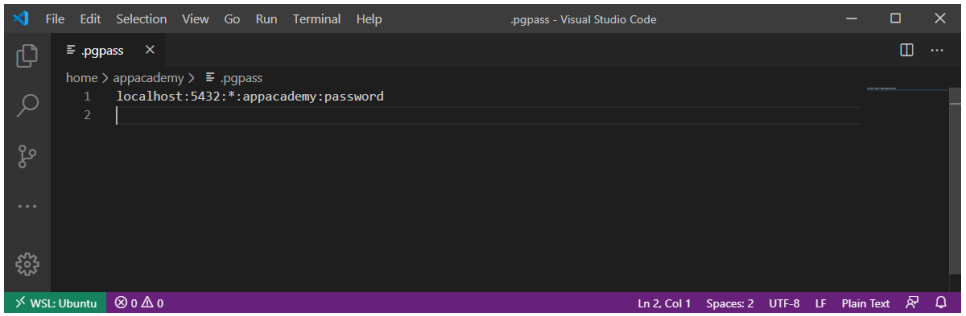
Now you will create your PostgreSQL password file. Type the following into your Ubuntu shell to open Visual Studio Code and create a new file.

```
code ~/.pgpass
```

In that file, you will add this line, which tells it that on localhost for port 5432 (where PostgreSQL is running), for all databases (*), **use your Ubuntu user name and the password that you just created for that user with the `psql` command you just typed.** (If you have forgotten your Ubuntu user name, type `whoami` on the command line.) Your entry in the file should have this format.

```
localhost:5432:*:«your Ubuntu user name»:«the password you just used»
```

For the curriculum writers' systems, it looks like this in Visual Studio Code.



Once you have that information in the file, save it, and close Visual Studio Code.

The last step you have to take is change the permission on that file so that it is only readable by your user. PostgreSQL will ignore it if just anyone can read and write to it. This is for *your* security. Change the file permissions so only you can read and write to it. You did this once upon a time. Here's the command.

```
chmod go-rw ~/.pgpass
```

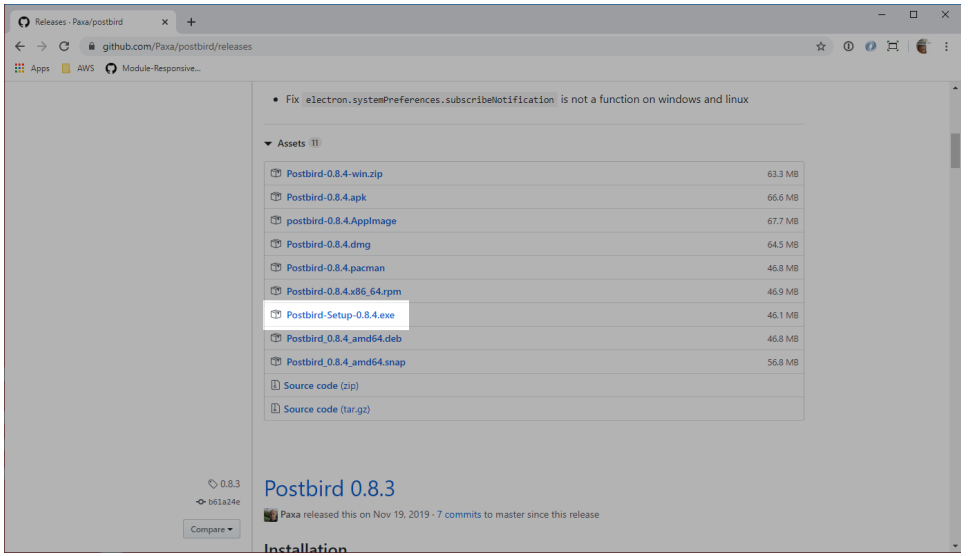
You can confirm that only you have read/write permission by typing `ls -al ~/.pgpass`. That should return output that looks like this, **with your Ubuntu user name instead of "appacademy".**

```
-rw------- 1 appacademy appacademy 37 Mar 28 21:20 /home/appacademy/.pgpass
```

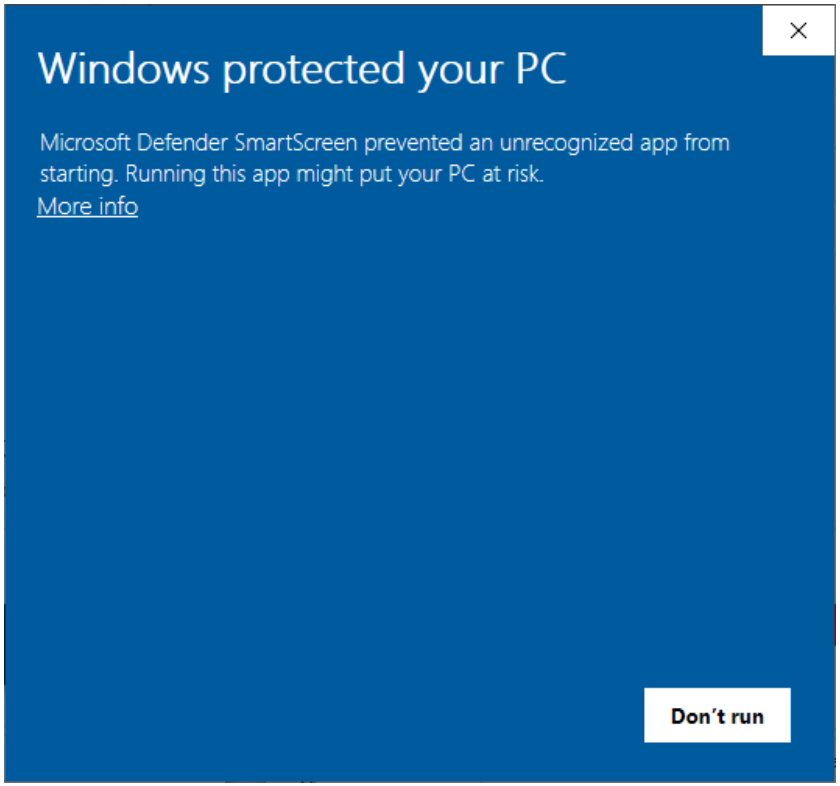
Now, try connecting to PostreSQL by typing `psql postgres`. Because you added the alias to your startup script, and because you created your **.pgpass** file, it should now connect without prompting you for any credentials! Type `\q` and press Enter to exit the PostgreSQL command line client.

Installing Postbird

Head over to the Postbird releases page on GitHub. Click the installer for Windows which you can recognize because it's the only file in the list that ends with ".exe".

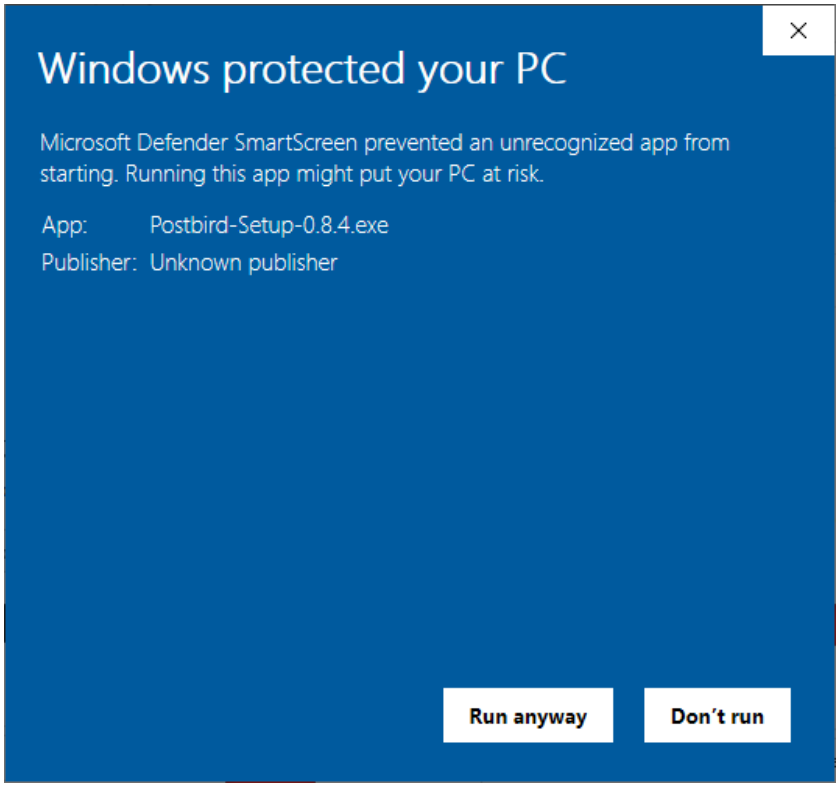


After that installer downloads, run it. You will get a warning from Windows that this is from an unidentified developer. If you don't want to install this, find a PostgreSQL GUI client that you do trust and install it or do everything from the command line.



You should get used to seeing this because many open-source applications aren't signed with the Microsoft Store for monetary and philosophical reasons.

Otherwise, if you trust Paxa like App Academy and tens of thousands of other developers do, then click the link that reads "More info" and the "Run anyway" button.



When it's done installing, it will launch itself. Test it out by typing the "postgres" into the "Username" field and the password from your installation in the "Password" field. Click the Connect button. It should properly connect to the running

You can close it for now. It also installed an icon on your desktop. You can launch it from there or your Start Menu at any time.

What you did

You installed and configured PosgreSQL 12, a relational database management system, and tools to use it! Well done!

Installing PostgreSQL 12 and Postbird on macOS

You will install two pieces of software so that you can start using PostgreSQL. You sill install PostgreSQL itself along with all of its tools. Then you will also install Postbird, a cross-platform graphical user interface that makes working with SQL and PostgreSQL better than just using the command line tool `psql`.

You can install both of these products using Homebrew. Your Windows-using classmates don't have this convenience, so pretend you're having a hard time doing this. 😉

Installing PostgreSQL 12

First, update your Homebrew installation. You should do this before each thing that you install using Homebrew.

```
brew update
```

Now, make sure that you have the correct Homebrew recipe. Type the following and make sure that the first line of the output contains some version of "12".

```
brew info postgresql
```

You should see something like this in the output.

```
postgresql: stable 12.2 (bottled), HEAD
```

Now, launch the installation.

```
brew install postgresql
```

This may take a while. Have a tea.

Configuring PostgreSQL 12

When that completes, you can read the **Caveats** section from the installation output. You should do this with *everything* that you install using Homebrew.

```
To migrate existing data from a previous major version of PostgreSQL run:
  brew postgresql-upgrade-database

To have launchd start postgresql now and restart at login:
  brew services start postgresql
Or, if you don't want/need a background service you can just run:
  pg_ctl -D /usr/local/var/postgres start
```

You definitely want PostgreSQL to run now and every time you log in. Otherwise you'd have to start it every time you reboot your computer which can be a hassle. Following the instructions, please type the following into the command line.

```
brew services start postgresql
```

That should report that PostgreSQL is now started.

Connecting to PostgreSQL

To make sure your client tools are configured properly, type the following in a Terminal. This tells the `psql` client that you want to connect to the "postgres" database, which is the default database created when PostgreSQL is installed.

```
psql postgres
```

That will connect to the "postgres" database as your user that you're logged in as, which the installer configured for you during the installation. It's the same as specifying your user name by using the "-U" command line parameter and typing.

```
psql -U «your user name» postgres
```

When you successfully log in, it should show you the following output.

```
psql (12.2)
Type "help" for help.

postgres=#
```

Type `\q` and hit Return to quit the PostgreSQL client.

Installing Postbird

Make sure that your Homebrew can find Postbird. Search for it using the `brew search` command.

```
brew search postbird
```

You should get something back that looks like this.

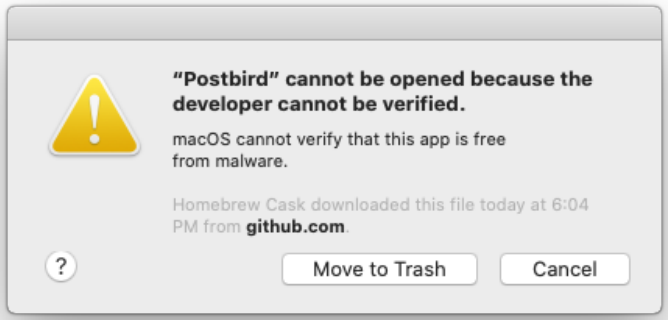
```
==> Casks
postbird
```

That means it could find it. Since it's a **Cask**, that means it's an application that is meant to be used as a graphical user interface rather than on the command line. To install it, you need to include "cask" in the command.

```
brew cask install postbird
```

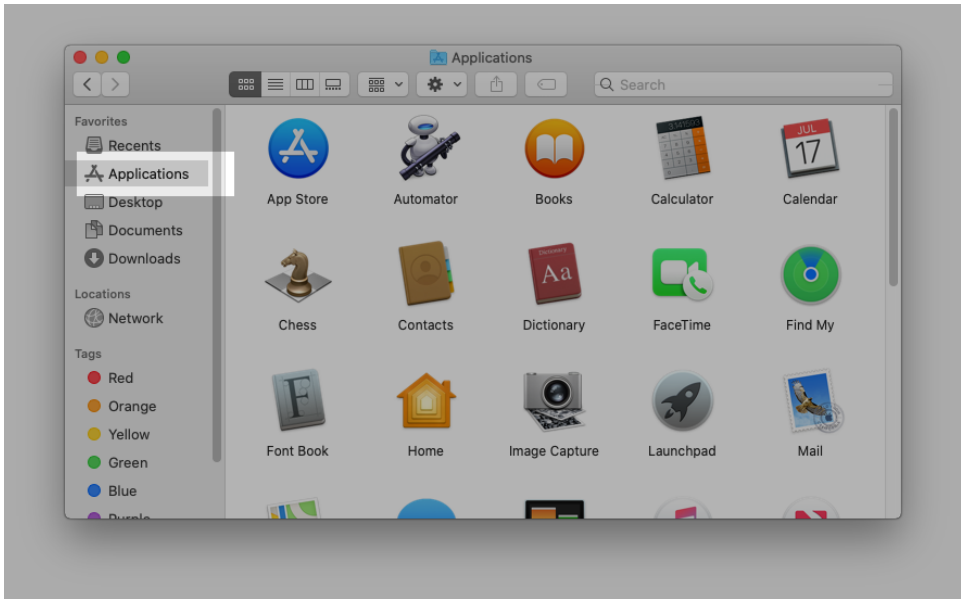
Once it installs, try starting it. It's in your Applications directory. You can use Spotlight to launch it by pressing Command+Space. In the Spotlight window, type "postbird". It should show you the recently-installed application's logo in the list as a white circle with a blue elephant. Select that and press Enter (or click it, if you're the touchpad/mouse kind of person).

The first time it starts, you may get this error.

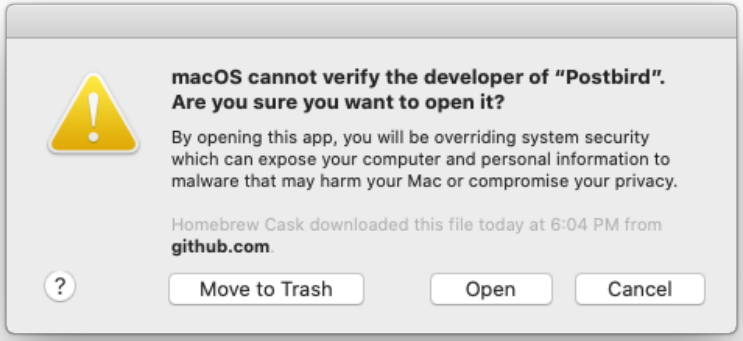


You should get used to seeing this because many open-source applications aren't signed with an Apple Developer Certificate yet. Click "Cancel". We will need to tell macOS we would like to run this application despite it not being signed.

Open up your Applications directory by clicking on the shortcut in the left bar of Finder.



Find the Postbird application in there. Hold down the Option key and left-click the icon. (If you're using the touchpad, this probably means two-finger tap.) Once the context menu shows up, you can let go of the Option key. Then, click the "Open" menu item. You will now see a new version of the popup from before with an extra button.



Click the "Open" button. That will let Postbird run. Now that you've done that once, you won't have to do it again for Postbird. This is valuable knowledge about how to run open-source software on macOS that isn't signed by the developers. However, it is a good security practice to only run applications from developers that you trust. The Postbird application is used by tens of thousands of software developers and is something that you can trust.

In fact, the developer of Postbird has an 'issue' on github about this very thing. So hopefully they will get the signing of the app fixed in a future version.

When Postbird starts, type "postgres" into the database field. Then, click the "Connect" button. It should open a new tab and show you basically a blank page. That means everything worked! Exit Postbird by pressing Command+Q or selecting Postbird > "Quit Postbird" from the menu.

What you did

You installed and configured PosgreSQL 12, a relational database management system, and tools to use it! Well done!

User Management Walk-Through

This is a walk-through: Please type along as you read what's going on in this article.

In this walk-through, you will

- Create superusers with full access to the system,
- Create normal users,
- Delete users from the system, and,
- See how SQL is not case sensitive

It's good practice to create a different database user for each application that you will have connect to an RDBMS. This prevents applications from reading and changing data in *other* databases that they should not have access to.

The "User" in PostgreSQL is a **database entity** that represents a person or system that will connect to the RDBMS and access data from one or more databases. This is the first entity that you will create, modify, and destroy.

All user management is beyond the scope of the ANSI SQL standard. That means each relational database management system has its own vendor-specific extensions about how to do this. When working with a new RDBMS, check out its documentation about how to create users, groups, and other security entities.

Naming a user

Names of users should not create spaces or dashes. They should contain only lower case letters, numbers, and underscores.

- Good user names
  - appacademy
  - patel_kush_112
  - bdorsey
- Bad (incorrect) user names
  - Ned Ruggeri
  - melvin-howard-tormé
  - b.d.o.r.s.e.y

Creating a superuser

On Windows, open your Ubuntu shell. On macOS, open your Terminal. Start the PostgreSQL command line client with the command `psql postgres`.

You should see some information about the version of the database and the command line tool, plus a helpful hint to type "help" if you need help. Then, you will see the `psql` prompt:

```
postgres=#
```

The value "postgres" means that you're currently connected to the "postgres" database. More on that in the next article.

To create a user in PostgreSQL you will use the following command. It creates a user with the name "test_superuser" and the password "test". *Type* that command (please don't copy and paste it) and run it by hitting Enter (or Return).

```
CREATE USER test_superuser
WITH
PASSWORD 'test'
SUPERUSER;
```

Note that this SQL statement ends with a semicolon. All SQL statements in PostgreSQL do. Don't forget it. If you do forget it, just type it on an empty line. The above statement, for example, can also be entered as the following where the semicolon is on a line all its own.

```
CREATE USER test_superuser
WITH
PASSWORD 'test'
SUPERUSER
;
```

If you typed it correctly, you will see the message `CREATE ROLE`. Because you created test_superuser as a super user, when a person or application uses that login, they can do whatever they want. You will test out that fact, now.

Quit your current session by typing `\q` and hitting Enter (or Return). Now type the following command to connect to PostgreSQL with the newly-created user. It instructs the client to connect as the user "test_superuser" (`-U test_superuser`) to the database named "postgres" (`postgres`) and prompt for the password (`-W`) for the user.

```
psql -W -U test_superuser postgres
```

At the prompt, type the password *test* that you used when you created the user. If everything went well, then you will find yourself at the SQL prompt just like before. To prove to you that you're now the "test_superuser", type the following command.

```
SELECT CURRENT_USER;
```

It should respond with the following output:

```
  current_user
---------------
 test_superuser
(1 row)
```

Creating a limited user

You're logged in as a super user that can do anything. Use that power! Create another user that does not have such amazing power. You will rarely create super users in real life. The following user creation is more appropriate. It creates just a normal user that can log in. Then, you can assign that user specific access to specific databases.

```
CREATE USER test_normal_user
WITH
PASSWORD 'test';
```

That should also give you the `CREATE ROLE` message that means everything went ok.

Quit the session by typing `\q` and pressing Enter (or Return). Start another as the new user.

```
psql -U test_normal_user -W postgres
```

Type the password *test* for this user. Confirm that you are now that new user by using the `SELECT CURRENT_USER;` command. Once confirmed, try to create a user named *hacking_the_planet* with a password of *pwned!*. What happens?

That's right. This user doesn't have the security privileges to create users.

Create users to do the job you want them to do. Then, give the appropriate permissions to that user. This will make a safe and secure application development world for you and your team.

Removing a user

Time to remove both of these users. The opposite of `CREATE USER` is `DROP USER`. To drop a user, you just type `DROP USER «user name»;`.

Connect again as just you, the OG super user. (Once again, that's with the command `psql postgres`.)

Drop the normal user with the command

```
DROP USER test_normal_user;
```

Then, drop the user with the name "test_superuser". You should receive the message "DROP ROLE" for each of your commands.

Case sensitivity

Unlike JavaScript, the keywords in SQL are case insensitive. that means you can type any of the following and they'll all work.

```
DROP USER test_user;
Drop User test_user;
drop user test_user;
```

Notice that entity names like user names *are* case sensitive.

SQL is conventionally written in all uppercase to distinguish the commands from the names that you will have for your entities and their properties.

What you learned

- That a user is a *database entity* in PostgreSQL
- That it is best practice to create a user for each application that you will create
- How to create a super user with the `CREATE USER ... SUPERUSER` command
- How to create a restricted (normal) user with the `CREATE USER` command
- How to remove a user with the `DROP USER` command
- SQL keywords are not sensitive to case, but are conventionally written in uppercase

---

Database Management Walk-Through

This is a walk-through: Please type along as you read what's going on in this article.

In this walk-through, you will

- Create a database for your user,
- Create databases for other users,
- Apply security to the databases to prevent access,
- See the first example of "relational data" in the RDBMS, and,
- Create other databases and apply security to them.

Now that you can create users for each of your applications, it's time for you to be able to create a **database**. The database is where you will store the data for your application.

You've been using the following command to log in as your superuser to the "postgres" database. This works because if you don't specify a user with the `-U` command line parameter, it just uses the name of the currently logged in user, your user name.

```
psql postgres
```

That's because if you don't specify a database, then PostgreSQL will try to connect you to a database that has the same name as your user. Try it, now.

```
psql
```

When you run this, if there is no database with your user name, then you will receive an error message that reads like the following. (The text has been wrapped in the example for readability.)

```
psql: error: could not connect to server:
      FATAL:  database "appacademy" does not exist
```

Naming a database

Names of databases should not create spaces or dashes. They should contain only lower case letters, numbers, and underscores.

- Good database names
  - appacademydata
  - financials2020
  - chicago_office
- Bad (incorrect) database names
  - App Academy Data
  - financials-2020
  - chicago.office

Creating a database for your user

So that you don't have to type "postgres" every time you want to connect on the command line, you can create a database with your user name so that one will exist. To determine your user name, type the following command at your shell, *not* in PostgreSQL.

```
whoami
```

That will show you the name of your user. Remember that name. Now, start your PostgreSQL command line as your superuser.

```
psql postgres
```

That should result in the `psql` command prompt of `postgres=#`. Again, that means that you are currently connected to the "postgres" database.

Once you're greeted by the `postgres=#` command prompt, you can create a database for your user by typing the following command. Don't copy and paste, here. Type it out.

```
CREATE DATABASE «your user name» WITH OWNER «your user name»;
```

By making yourself the owner of that database, then your user can do anything with it.

For the examples in these articles, the user name is "appacademy", so the authors typed

```
CREATE DATABASE appacademy WITH OWNER appacademy;
```

If the command succeeds, you will see the message "CREATE DATABASE". Now, quit the client using `\q`. Now, connect, again, to PostgreSQL by just typing the

following.

```
psql
```

Now, when you log in, you will be greeted by a command prompt that reads

```
«your user name»=#
```

You're connected to your very own database! And, now, you have less to type when you want to start `psql`! Yay for less typing!

Creating other users and databases

Create two normal users with the following user names and passwords using the `CREATE USER` command from the last article.

User name	Password
ada	ada1234
odo	ODO!!!1

Now, create two databases, each named for a user with that user as the owner. Again, type these rather than copying and pasting.

```
CREATE DATABASE ada WITH OWNER ada;
CREATE DATABASE odo WITH OWNER odo;
```

Now that you have new users and databases for them, it's time to test out that you can connect to PostgreSQL with those users. Quit your current session by typing `\q` and pressing Enter (or Return). Then, start a new session for "ada" by using the following `psql` command that will prompt for the user's password (`-W`) and connect as the specified user (`-U ada`).

```
psql -W -U ada
```

Note: Those command line parameters can come in any order, usually. The above statement can also be written as `psql -U ada -W`, for example.

When you enter that and type the password for "ada" which is "ada1234" from the table above, you should see that you are now connected to the "ada" database in the prompt.

```
ada=>
```

Notice that the character at the end is now a ">" rather than the "#" that you're used to seeing. That's because "ada" is a normal user. Normal user prompts end with ">". Your user, the one tied to your user name, is a super user. That results in a "#"

Quit this session and connect as the "odo" user, now. You will notice that because "odo" is a normal user, that you will see this prompt, too.

```
odo=>
```

Applying security to databases

You've created a database for "odo". Type the following command which will try to connect as the user "ada" (`-U ada`) to the database "odo" (`odo`).

```
psql -W -U ada odo
```

After you type the password, you may be surprised to find out that "ada" can connect to the database "odo" that's owned by the user "odo"! That's because all databases, when they're created, by default allow access to what is known as the "PUBLIC" schema, a kind of group that everyone belongs to. You sure don't want

that if you want to prevent the user "ada" from messing up the data in the database "odo", and the user "odo" from messing up the data in the database "ada".

To do that, you have to revoke connection privileges to "PUBLIC". That's like putting a biometric lock on a bank safety deposit box so that only the owner of that deposit box (and bank officials) can get into it and do stuff with its contents.

To do that, quit the current `psql` session if you're in one. Connect to PostgreSQL as your user, a superuser. Again, now that you have your own database, you can just type `psql` at your macOS Terminal command line or Ubuntu shell command line. Once you have your prompt

```
«your user name»=#
```

you want to type a command that will revoke all privileges from the databases named "odo" and "ada" the connection privileges of the entire "PUBLIC" group. To do that, you write the command with the form:

```
REVOKE CONNECT ON DATABASE «database name» FROM PUBLIC;
```

Do that for both databases. Each time you run it, you should see the message "REVOKE" showing that it worked.

Now, quit your session (`\q`). With the connection privilege revoked, "ada" can no longer connect to database "odo" and vice versa. Try typing the following.

```
psql -W -U ada odo
```

You should see an error message similar to the following.

```
psql: error: could not connect to server:
      FATAL:  permission denied for database "odo"
DETAIL:  User does not have CONNECT privilege.
```

Try connecting with the user "odo" to the database named "ada". You should see the same error message except with the database named "ada" in it.

But, your superuser status will not be thwarted! You can still connect to either of those because of your superuser privileges. Neither of the following commands should fail.

```
# Connect to the database "odo" as your superuser
psql odo
```

```
# Connect to the database "ada" as your superuser
psql ada
```

Superusers can connect to any and all databases. Because *superuser*!

Remember you created a database for your user? Now, revoke connection privileges from it for "PUBLIC", too.

Listing databases and granting privileges

Now, say the user "ada" needs another database, one that will contain data that "ada" wants to keep separate from the data in the database "ada". Connect to PostgreSQL as your user. Create a new database without specifying the owner.

```
CREATE DATABASE hr_data;
```

Now, type the command to list databases on your installation of PostgreSQL.

```
\list
```

You will see something akin to the following. The entries in the "Collate", "Cypte", and "Access privileges" columns may differ. That's fine and can be ignored. Also, where you see "appacademy", you'll probably see your user name.

Name	Owner	Encoding	Collate	Ctype	Access p
ada	ada	UTF8	C	C	=T/ada +
					ada=CTc/ada
appacademy	appacademy	UTF8	C	C	
hr_data	appacademy	UTF8	C	C	
odo	odo	UTF8	C	C	=T/odo +
					odo=CTc/odo
postgres	appacademy	UTF8	C	C	
template0	appacademy	UTF8	C	C	=c/appacademy
					appacademy=C
template1	appacademy	UTF8	C	C	=c/appacademy
					appacademy=C

You will see that for the database that you just created, "hr_data", that the owner is you. Go ahead and revoke all access to it from "PUBLIC" like you did in the last section. Once you've done that, no one but you (and other superusers) can connect to the "hr_data" database. (You may want to exit the `psql` shell and try connecting with the credentials for the "ada" user just to make sure. If you do that, reconnect as your user so you can continue with the security management.)

Now, you need to add "ada" back to it so that user can connect to the database. The opposite of `REVOKE ... FROM ...` is `GRANT ... TO ...`. So, you will type the following:

```
GRANT CONNECT ON DATABASE hr_data TO ada;
```

Now, if you exit the `psql` shell and connect as "ada", you will see that user can connect. Make sure that's true.

```
psql -U ada hr_data
```

Once you have confirmed that "ada" can connect, make sure that user "odo" cannot connect.

```
psql -U odo hr_data
```

That command should return the error message that reads that the user "does not have CONNECT privilege."

Time to clean up

Time to clean up the entities that you've created in this walk-through. You already know how to delete a user by using the `DROP USER` statement. Log in as your superuser and try to drop the "ada" user. You should see an error message similar to the following.

```
ERROR:  role "ada" cannot be dropped because some objects depend on it
DETAIL:  owner of database ada
privileges for database hr_data
```

This tells you that you can't drop that user because database objects in the system rely on the existence of the user "ada". This is the first example that you've seen of *relational data*. The database "ada" is related to the user "ada" because user "ada" owns the database "ada". The database "hr_data" is

related to the user "ada" because the user "ada" has access privileges for the database "hr_data".

This is one of the primary reasons that relational databases provide such an important role in application design and development. If you or your application puts data into the database that relates to other data, you can't just remove it without removing *all of the related data, too*!

To remove the related data from user "ada", you need to revoke the connect privilege on "hr_data" for user "ada". Then, you need to delete the database "ada" that user "ada" owns. You've seen some `REVOKE` statements in this article that revoke the connect privilege from "PUBLIC". It's the same for an individual user, too, just replace "PUBLIC" with the name of the user.

Then, the opposite of `CREATE DATABASE` is `DROP DATABASE` just like the opposite of `CREATE USER` is `DROP USER`.

Putting together those two hints, you can type commands like this to get the job done.

```
REVOKE CONNECT ON DATABASE hr_data FROM ada;
DROP DATABASE ada;
DROP USER ada;
```

Run in that order, the first two statements remove the data *related* to the user "ada". Once that's gone, you can finally remove the user "ada" itself.

Do the same for the user "odo", deleting the related data, first. Remember, you can run the `DROP` statement for the user "odo" to see what data relates to that user.

Note: When you run a statement in PostgreSQL that results in an error message, do not worry! You have not corrupted anything! These are helpful statements to let you know that the state of the database won't allow you to perform the requested operation. These kinds of error statements are guideposts for you to follow to get to the place you want to be.

What you've done

You have successfully created databases for yourself and other users. You have created a database with you as the owner and given access to it to another user. You have locked down databases so only owners (and superusers) can access them. You know how to see the owner of a database. You know how to remove a user from a database after removing all data related to the user.

This is the start of a lovely secure set of databases.

Table Management Walk-Through - Part I

This is a walk-through: Please type along as you read what's going on in this article.

In this walk-through, you will

- Learn about what a table is,
- How to create and delete tables,
- Who owns a table, and,
- Learn about different data types that you can use when defining a table.

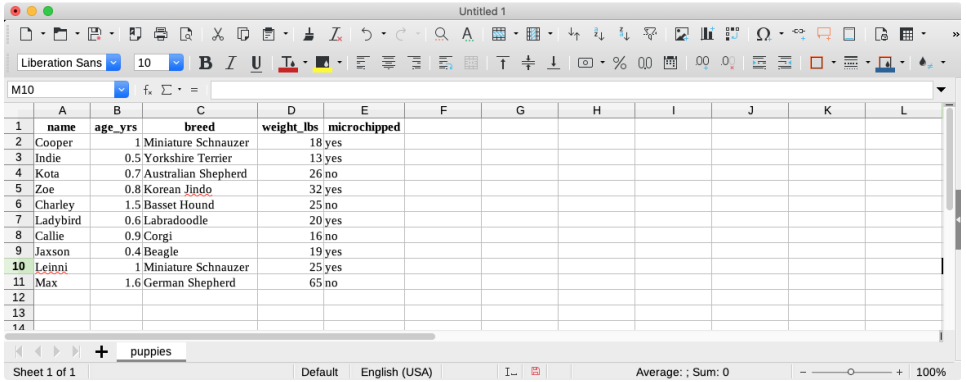
You can now create users that can connect to the relational database management system. You can now create databases and secure them so only certain users can connect to them. Now, it's time to get into the part where you define the entities that actually hold the data: **tables**!

What is a table?

A table is a "logical" structure that defines how data is stored and contains that data that meets the definition. Most people think about tables like spreadsheets that have a specific number of columns and rows that contain the data.

It is called a "logical" structure because we reason about it in terms of columns and rows; however, the RDMBS is in charge of how the data is actually stored on disk and, quite often, for performance reasons, it does *not* look like rows and columns. The way it is stored on disk is called the "physical" structure because that's what is the actual physical representation of it. We do not cover physical structures because they vary by RDBMS. If you become a **database administrator** in the future, you may have to learn such things.

Here is a spreadsheet that contains some data about puppies.



	A	B	C	D	E	F	G	H	I	J	K	L
1	name	age_yrs	breed	weight_lbs	microchipped							
2	Cooper	1	Miniature Schnauzer	18	yes							
3	Indie	0.5	Yorkshire Terrier	13	yes							
4	Kota	0.7	Australian Shepherd	26	no							
5	Zoe	0.8	Korean Jindo	32	yes							
6	Charley	1.5	Basset Hound	25	no							
7	Ladybird	0.6	Labradoodle	20	yes							
8	Callie	0.9	Corgi	16	no							
9	Jaxson	0.4	Beagle	19	yes							
10	Leinni	1	Miniature Schnauzer	25	yes							
11	Max	1.6	German Shepherd	65	no							
12												
13												

You can see that the columns are

- name
- age_yrs
- breed
- weight_lbs
- microchipped

Now, look at the data each column contains. You can guess at what kind of data type is in each of them by their values. If you were to write that out using the data types that you know from JavaScript, you might come up with the following table.

Column	Data type
name	string
age_yrs	number
breed	string
weight_lbs	number
microchipped	Boolean

In table definitions, you have to be more specific, unfortunately. This is so the database can know things like "the maximum length of the string" or "will the number have decimal points"? This is important information so that database can know how to store it most efficiently. The following table shows you the corresponding ANSI SQL data types for the JavaScript types from before.

Column	JavaScript data type	Max length	ANSI SQL data type
name	string	50	VARCHAR(50)
age_yrs	number		NUMERIC(3,1)
breed	string	100	VARCHAR(100)
weight_lbs	number		INTEGER
microchipped	Boolean		BOOLEAN

You can see that "string" values map to something called a "VARCHAR" with a maximum length.

You can see that "number" values map to something called a "NUMERIC" with some numbers, or an INTEGER which is just a whole number.

You can see that "Boolean" values map to something called a "BOOLEAN" which is nice because that's convenient.

Defining tables

To define a table, you need to know what the different pieces of related data it will store. Then, you need to know what kind each of those pieces are. Once you have that, you can create a table with an ANSI SQL statement.

String types

There are three kinds of commonly used string types that databases support based on the ANSI SQL standard. This section talks about them.

The most commonly used type is known as the **CHARACTER VARYING** type. It means
that it can contain text of varying lengths up to a specified maximum. That maximum is provided when you define the table. Instead of having to type *CHARACTER VARYING* all the time, you can use its weirdly named alias **VARCHAR**, (pronounced "var-car" or "var-char" where each part rhymes with "car"). So, to specify that a column can hold up to 50 characters, you would write `VARCHAR(50)` in the table definition. (Remember, SQL is case insensitive for its keywords. You can also write `varchar(50)` or `VarChar(50)` if you so desired. Just be consistent.)

Another commonly used type is known simply as **TEXT**. This is a column that can contain an "unlimited" number of characters. You may ask yourself, "Why don't I just always use TEXT, then?" Performance is the reason. Columns with the *TEXT* type are notoriously slower than those with other string types. Use them judiciously.

Purposefully left out from this discussion is a type named **CHARACTER** or **CHAR**. It is like the **VARCHAR**, except that it is a fixed-width character field. This author has *never* seen it used in a production system except for Oracle DB which did not, at one time, support a Boolean type. Other than that, it was only useful back in the 1970s - 1990s when computer disk space and speed were slow and expensive.

Numeric types

ANSI SQL (and PostgreSQL) supports **INTEGER**, **NUMERIC**, and floating-point numbers.

The *INTEGER* should be familiar. It's just a number. In PostgreSQL, it can hold almost all values that your application can handle. That's from -2,147,483,648 to +2,147,483,647. If, for some reason, you were writing a database that would contain a record for every person in the world, you would need integers bigger than that. To solve that problem, ANSI SQL (and PostgreSQL) supports the **BIGINT** type that will hold values between -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. If your application needs bigger integers, there are extensions available.

The **NUMERIC** type is a fixed-point number. When you specify it, it takes up to two arguments. The first number is the total number of digits that a number can have in that column. The second number is the number of digits after the decimal point that you want to track. The specification *NUMERIC(4,2)* will hold the number *23.22*, but not the numbers *123.22* (too many total digits) or *23.222* (which it will just ignore the extra decimal places and store *23.22*). These exact numbers are important for things like storing values of money, where rounding errors could cause significant errors in calculations.

If you don't care about rounding errors, you can use the **DOUBLE PRECISION**. There is no short alias for it. You can just put decimal numbers in there and they will come out pretty much the same. Don't use this kind of data type for columns that contain values of money because they will round and someone will get in trouble, eventually.

Other data types

PostgreSQL supports a lot of other data types, as well. It has specialized data types for money, dates and times, geometric types, network addresses, and JSON! Ones that you will use a lot in this course are the ones for dates and times, as well as the one for JSON.

Here's the link to the documentation on PostgreSQL data types. Go review the documentation for the types that support dates and times as you will need to know the **TIMESTAMP** and **DATE** types.

Naming a table

Names of tables should not create spaces or dashes. They should contain only lower case letters, numbers, and underscores.

Conventionally, many software developers name their database table names as the plural form of the data that it holds. More importantly, many software libraries known as ORMs (which you will cover, this week) use the plural naming convention. You should use the plural naming convention while here at App Academy.

- Good table names
  - student_grades
  - office_locations
  - people
- Bad (incorrect) table names
  - Student Grades
  - office-locations
  - person

Note: The opinion that database table names should be plural is the subject of heated debate among many software developers. We don't argue about it at App Academy. We acknowledge that it *really doesn't matter* how they're named. You should just be consistent in the way they're named. Because our tools will use the plural forms, we use the plural forms.

Writing the SQL

Creating a table with SQL has this general syntax.

```
CREATE TABLE «table name» (
  «column name» «data type»,
  «column name» «data type»,
  ...
  «column name» «data type»
);
```

A couple of things to note. First, it uses parentheses, not curly braces. Many developers that use curly brace languages like JavaScript will eventually, out of habit, put curly braces instead of parentheses. If you were to do that, the RDBMS will tell you that you have a syntax error. Just grin and replace the curly braces with parentheses.

Another thing to note is that the last column specification *cannot* have a comma after it. In JavaScript, we can have commas after the last entry in an array or in a literal object definition. Not so in SQL. Again, the RDBMS will tell you that there is a syntax error. Just delete that last comma.

Here's the table that contains the column definitions for the "puppies" spreadsheet from before.

Column	JavaScript data type	Max length	ANSI SQL data type
name	string	50	VARCHAR(50)
age_yrs	number		NUMERIC(3,1)
breed	string	100	VARCHAR(100)
weight_lbs	number		INTEGER
microchipped	Boolean		BOOLEAN

To write that as SQL, you would just put in the table name, column names, and data types in the syntax from above. You would get the following.

```
CREATE TABLE puppies (
  name VARCHAR(50),
  age_yrs NUMERIC(3,1),
  breed VARCHAR(100),
  weight_lbs INTEGER,
  microchipped BOOLEAN
);
```

Log into your database, if you're not already. (Make sure you're in *your* database by looking at the prompt. It should read `«your user name»=#`.) Type in the SQL statement from above. If you do it correctly, PostgreSQL will return the message "CREATE TABLE".

Listing tables and table definitions

You can see the tables in your database by typing `\dt` at the `psql` shell prompt. The `\dt` command means "describe tables". If you do that now, assuming that you've only created the "puppies" table, you should see the following with your user name, of course.

```
          List of relations
 Schema |  Name   | Type  |   Owner
--------+---------+-------+------------
 public | puppies | table | appacademy
```

The user that runs the SQL statement that creates the table is the owner of that table. Table owners, like database owners, will always be able to access the table and its data. If you want a user other than the one that you're logged in as to own the table, you have two ways of doing that.

- Log out and log in as the user that you want to own the table and run the `CREATE TABLE` statement as that user.
- As the superuser, run the `SET ROLE «user name»` command to switch the current user and run the `CREATE TABLE` statement as that user.

To see the definition of a particular table table, type `\d «table name»`. For puppies, type `\d puppies`. You should see the following output.

```
                      Table "public.puppies"
    Column    |          Type          | Collation | Nullable | Default
--------------+------------------------+-----------+----------+---------
 name         | character varying(50)  |           |          |
 age_yrs      | numeric(3,1)           |           |          |
 breed        | character varying(100) |           |          |
 weight_lbs   | integer                |           |          |
 microchipped | boolean                |           |          |
```

For now, ignore the "Collation", "Nullable", and "Default" columns in that output. The next article will address "Nullable" and "Default".

You can see that the data types that you provided have been translated into their ANSI SQL full name equivalents.

Now, connect to the "postgres" database using the `\c postgres` command. It should give you a message that you're now connected to the "postgres" database as your user. The prompt should change from one that has your name to `postgres=#`. Now, type `\dt` to list the tables in the "postgres" database. If you haven't created any tables there, it will read "Did not find any relations." If you type `\d puppies`, it will report that it can't find a relation named "puppies".

This is because you're in a different database than the one in which you created the "puppies" table. You just don't see the "puppies" table, here, because it doesn't exit. That table is in another database, your user database. That's how databases work: they provide an area where you can create tables in which you'll store data. Different databases have different tables. You can't easily get at tables in another database from the one that you're currently in. And, really, you don't want to. Databases provide the storage and security boundaries for data.

Change back to your user database by executing the command `\c «your user name»`.

Deleting a table

In the same way that you can delete users and databases by using the `DROP` command, you can do the same for tables. To get rid of the "puppies" table, execute the following SQL command.

```
DROP TABLE puppies;
```

It should tell you that it dropped the table. You can confirm that it is no longer there by executing the `\dt` command.

What you've done

In this section, you learned the basics about creating database entities called "tables" and their ownership. You learned that tables are where you store data. You discovered that the data that you store is defined by the columns and their data types. You can now write SQL to create and drop tables.

Next up, you will learn about special kinds of columns, column constraints, and building relations between tables.

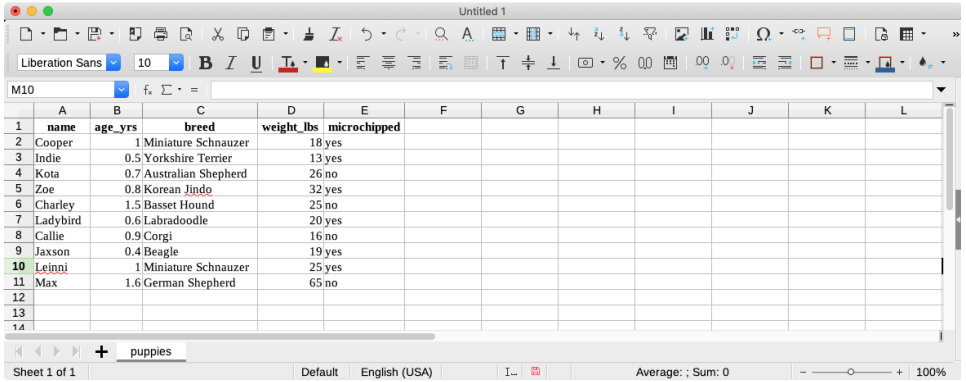
Table Management Walk-Through - Part II

This is a walk-through: Please type along as you read what's going on in this article.

In this walk-through, you will

- Learn about nullable columns,
- Learn about default values for columns,
- Learn how to make columns have unique values,
- Learn about primary keys, and,
- Learn about relating tables through foreign keys to maintain data and referential integrity.

Here is the "puppies" spreadsheet, table definition, and the SQL to create it from the last article.



name	age_yrs	breed	weight_lbs	microchipped
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

Column	JavaScript data type	Max length	ANSI SQL data type
name	string	50	VARCHAR(50)
age_yrs	number		NUMERIC(3,1)
breed	string	100	VARCHAR(100)
weight_lbs	number		INTEGER
microchipped	Boolean		BOOLEAN

```
CREATE TABLE puppies (
  name VARCHAR(50),
  age_yrs NUMERIC(3,1),
  breed VARCHAR(100),
  weight_lbs INTEGER,
  microchipped BOOLEAN
);
```

In this article, you will add more specifications to this table so that you can properly use it. Then, you will refactor it into two tables that relate to one another.

Nullable columns

By default, when you define a table, each column does not require a value when you create a record (row). Look at the spreadsheet. You can see all of the rows in it have data in every column. The SQL that you wrote does not enforce that.

The value `NULL` is a strange value because it means *the absence of a value*. When a value in a row is `NULL`, that means that it didn't get entered. Many database administrators, experts in databases and the models of data in them, detest the value `NULL` for one reason: it adds a weird state.

Think about a Boolean value in JavaScript. It can one of two values: `true` or `false`. In databases, a "nullable" `BOOLEAN` column, that is a `BOOLEAN` column that can hold `NULL` values, can have *three* values in it: `TRUE`, `FALSE`, and `NULL`. What does that mean to you as a software developer? It is this weird third state that leads to a strange offshoot of mathematics named three-valued logic. To prevent that, you should (nearly) always put the `NOT NULL` constraint on each of your column definitions. That will make your previous SQL statement look like this.

```
CREATE TABLE puppies (
  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
);
```

Type that SQL into your `psql` shell and execute it. (If you already have a "puppies" table, drop the existing one first.) Then, run `\d puppies`. You will see, now, that the column "Nullable" reads "not null" for every single one.

```
                    Table "public.puppies"
    Column    |          Type          | Collation | Nullable | Default
--------------+------------------------+-----------+----------+---------
 name         | character varying(50)  |           | not null |
 age_yrs      | numeric(3,1)           |           | not null |
 breed        | character varying(100) |           | not null |
 weight_lbs   | integer                |           | not null |
 microchipped | boolean                |           | not null |
```

Now, when someone tries to add data to the table, they must provide a value for every single column.

Note: An empty string is *not* a `NULL` value. It is still possible for someone to insert the string "" into the "name" column, for example. There are ways to prevent that, but you should check it in your JavaScript code before actually inserting the data.

Default values

Sometimes, you just want a column to have a default value. When there is a default value, the applications that insert data into the table can just rely on the default value and not have to specify it.

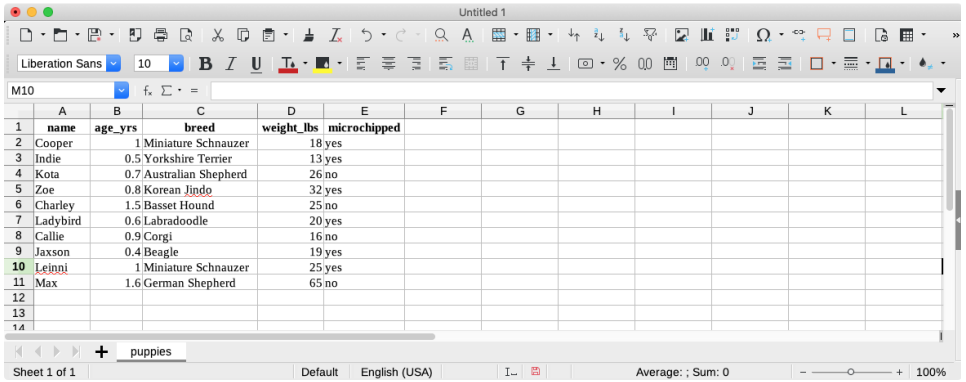
For the "puppies" table, a reasonable default value for the "microchipped" column would be `FALSE`. You can add that to your SQL using the `DEFAULT` keyword.

```
CREATE TABLE puppies (
  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
);
```

Drop the existing "puppies" table and type in that SQL. Then, run `\d puppies` to see how it shows up in the table definition.

Primary keys

Being able to identify a single row in a table is *very* important. Here's the screenshot of the spreadsheet, again.

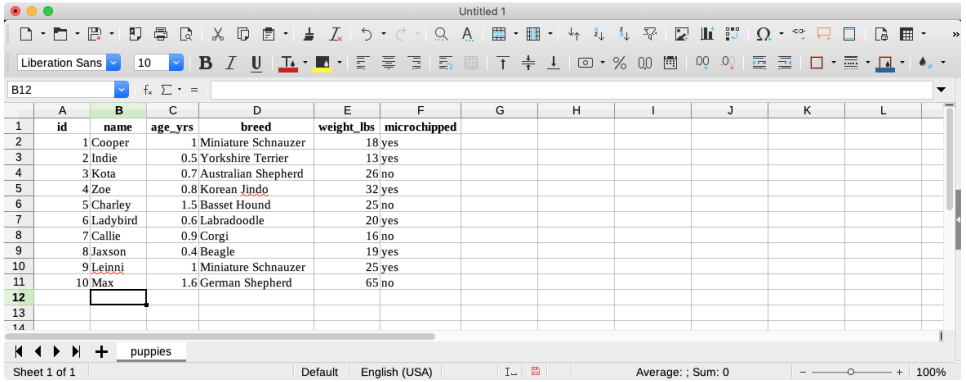


	A	B	C	D	E
1	name	age_yrs	breed	weight_lbs	microchipped
2	Cooper	1	Miniature Schnauzer	18	yes
3	Indie	0.5	Yorkshire Terrier	13	yes
4	Kota	0.7	Australian Shepherd	26	no
5	Zoe	0.8	Korean Jindo	32	yes
6	Charley	1.5	Basset Hound	25	no
7	Ladybird	0.6	Labradoodle	20	yes
8	Callie	0.9	Corgi	16	no
9	Jaxson	0.4	Beagle	19	yes
10	Leinni	1	Miniature Schnauzer	25	yes
11	Max	1.6	German Shepherd	65	no
12					
13					
14					

Let's say that the puppy named "Max" gains a couple of pounds. You want to update the spreadsheet. You scan through the list of names and find it on row 11. Then, you update the weight to be 69 pounds.

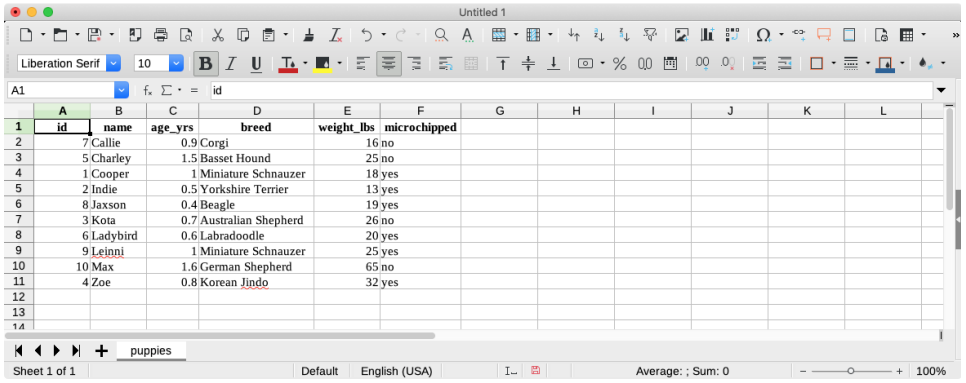
Now, what happens when you are tracking 300 dogs in the spreadsheet? What happens when your spreadsheet has 17 dogs named "Max"? It is helpful to have some way to uniquely identify a row in the spreadsheet. This is the idea behind a **primary key**. You can specify a column to be the primary key with the keywords `PRIMARY KEY`. A column that acts as a primary key cannot be `NULL`, so that is implied.

Here's the spreadsheet with a new column in it named "id" that just contains numbers to uniquely identify each row.



	A	B	C	D	E	F	G	H	I	J	K	L
1	id	name	age_yrs	breed	weight_lbs	microchipped						
2	1	Cooper	1	Miniature Schnauzer	18	yes						
3	2	Indie	0.5	Yorkshire Terrier	13	yes						
4	3	Kota	0.7	Australian Shepherd	26	no						
5	4	Zoe	0.8	Korean Jindo	32	yes						
6	5	Charley	1.5	Basset Hound	25	no						
7	6	Ladybird	0.6	Labradoodle	20	yes						
8	7	Callie	0.9	Corgi	16	no						
9	8	Jaxson	0.4	Beagle	19	yes						
10	9	Leinni	1	Miniature Schnauzer	25	yes						
11	10	Max	1.6	German Shepherd	65	no						
12												
13												

You may ask yourself, "Why can't I just use the row number as each row's identifier?" That's a very valid question! Here is the reason why. You can see that "Max" has an "id" of 10 on row 11. What happens if you wanted to look at the data differently, say sorted by name? Here's what that spreadsheet looks like.



	A	B	C	D	E	F	G	H	I	J	K	L
1	id	name	age_yrs	breed	weight_lbs	microchipped						
2	7	Callie	0.9	Corgi	16	no						
3	5	Charley	1.5	Basset Hound	25	no						
4	1	Cooper	1	Miniature Schnauzer	18	yes						
5	2	Indie	0.5	Yorkshire Terrier	13	yes						
6	8	Jaxson	0.4	Beagle	19	yes						
7	3	Kota	0.7	Australian Shepherd	26	no						
8	6	Ladybird	0.6	Labradoodle	20	yes						
9	9	Leinni	1	Miniature Schnauzer	25	yes						
10	10	Max	1.6	German Shepherd	65	no						
11	4	Zoe	0.8	Korean Jindo	32	yes						
12												
13												

You can see that when you sort them by name, if you relied on row number, "Max" now lives on row 10 rather than row 11. That changes the unique identifer of "Max" based on the way that you view the data. You want the unique identifier to *be part of the row definition* so that the number always stays with the row no matter how you've sorted the data. You will always know that the row with "id" value of 10 is "Max".

Keeping track of what the next number would be in that column could cause you a lot or headaches. What if two people (or applications) were entering data at the same time? Who would get the correct "next id" and still have it be unique? The answer to that is to let the database handle it. All databases have some way of specifying that you want to set the column to a special data type that will auto-assign and auto-increment an integer value for the column. In PostgreSQL, that special data type is called `SERIAL`.

Putting that all together, you would add a new column definition to your table with the name of "id" and the type `SERIAL`. Then, to specify that it is the primary key, you can do it one of two ways. The following example shows it as part of the column definition.

```
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
);
```

Or, you can put it in what is known as **constraint syntax** after the columns specifications but before the close parenthesis.

```
CREATE TABLE puppies (
  id SERIAL,
  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,
  PRIMARY KEY(id)
);
```

Either way you do it, when you view the output of `\d puppies`, you see some new things in the output.

```
                                  Table "public.puppies"
    Column    |          Type          | Collation | Nullable |              Default
--------------+------------------------+-----------+----------+-----------------------------
 id           | integer                |           | not null | nextval('puppies_id_seq'::regcl
 name         | character varying(50)  |           | not null |
 age_yrs      | numeric(3,1)           |           | not null |
 breed        | character varying(100) |           | not null |
 weight_lbs   | integer                |           | not null |
 microchipped | boolean                |           | not null | false
Indexes:
    "puppies_pkey" PRIMARY KEY, btree (id)
```

First, you'll notice that there is a weird default value for the "id" column. That's the way that PostgreSQL populates it with a new integer value every time you add a new row.

You will also see that that there is a section named "Indexes" after the column specifications. This shows that there is a thing named "puppies_pkey" which is the primary key on the column "id".

Unique values

Sometimes, you want all of the data in a column to be unique. For example, if you a table of people records. You want to collect their email address for them to sign up for your Web site. In general, people don't share email addresses (although it has been known to happen). You can put a constraint on a column by putting `UNIQUE` in the column's definition. For example, here's a sample "people" table with a unique constraint on the email column.

```
CREATE TABLE people (
  id SERIAL,
  first_name VARCHAR(50) NOT NULL,
  last_name VARCHAR(50) NOT NULL,
  email VARCHAR(250) NOT NULL UNIQUE,
  PRIMARY KEY (id)
);
```

When you use the `\d people` command to view the definition of the table, you will see this.

```
                                   Table "public.people"
   Column   |          Type          | Collation | Nullable |              Default
------------+------------------------+-----------+----------+-----------------------------------
 id         | integer                |           | not null | nextval('people_id_seq'::regclass
 first_name | character varying(50)  |           | not null |
 last_name  | character varying(50)  |           | not null |
 email      | character varying(250) |           | not null |
Indexes:
    "people_pkey" PRIMARY KEY, btree (id)
    "people_email_key" UNIQUE CONSTRAINT, btree (email)
```

Down there at the bottom, you see that PostgreSQL has added a `UNIQUE CONSTRAINT` to the list of indexes for the "email" field. Now, if someone tried to put an email address into the table that someone had previously used, then the database would return an error.

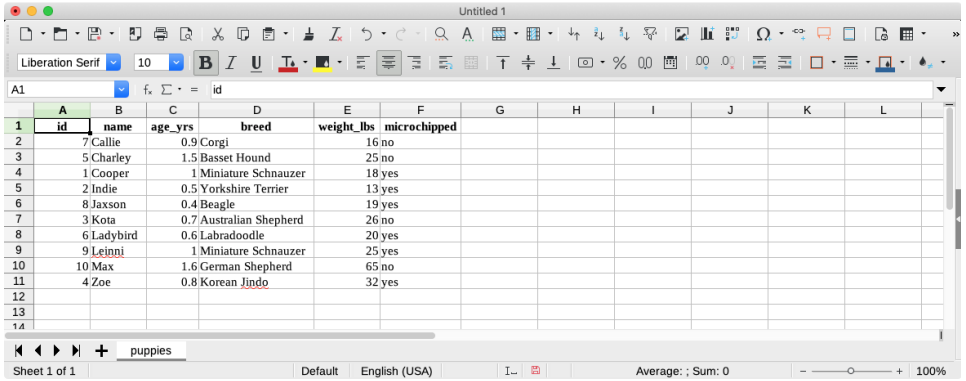
```
ERROR:  duplicate key value violates unique constraint "people_email_key"
DETAIL:  Key (email)=(a) already exists.
```

Refactor for data integrity

Now is the time for thinking about the nature of the data. When you create database tables, you need to ask yourself about the data that you're going to

store in them. One of the first questions that you should ask yourself is, "Do any of the columns have values that come from a list?" Or, another way to ask that is, "Do any of the columns come from a set of predefined values?" If you look at this data, does anything seem like it comes from a list, or that the data could repeat itself?

Take a look, again, at the spreadsheet. Does anything jump out at you?



id	name	age_yrs	breed	weight_lbs	microchipped
7	Callie	0.9	Corgi	16	no
5	Charley	1.5	Basset Hound	25	no
1	Cooper	1	Miniature Schnauzer	18	yes
2	Indie	0.5	Yorkshire Terrier	13	yes
8	Jaxson	0.4	Beagle	19	yes
3	Kota	0.7	Australian Shepherd	26	no
6	Ladybird	0.6	Labradoodle	20	yes
9	Leinni	1	Miniature Schnauzer	25	yes
10	Max	1.6	German Shepherd	65	no
4	Zoe	0.8	Korean Jindo	32	yes

If you looked at it and answered "the breed column", that's the ticket! The values that go into the breed column is finite. You don't want one person typing "Corgi" and another person typing "CORGI" and another "corgi" because, as you know, those are *three different values*! You want them all to be the same value! Supporting this is the primary reason that relational databases exist.

Instead of having just one table, you could have two tables. One that contains the puppy information and another that contains the breed information. Then, using the magic of relational databases, you can create a relation between the two tables so that the "puppies" table will reference entries in the "breeds" table.

This process is called **normalization**. It's a *really big deal* in database communities. And, it's a really big deal for application developers to maintain the integrity of the data. Bad data leads to bad applications.

To do this follows a fairly simple set of steps.

1. Figure out what related data repeats itself. In this case, it is only the single column that contains the **breed** names.
2. Create a new table to hold that data. Make sure it has a primary key. In this case, you can create a "breeds" table that contains an "id" the name of the breed.
3. Replace all of the columns in the original table that you extracted with a single value that will contain the corresponding "id" value from the new table. In this case, you will replace the "breed" column with a column named "breed_id" because it will have the id of the specific breed from the "breeds" table.

Here's what that would look like with two spreadsheets.

![Puppies and breed spreadsheets normalized]

You might think to yourself, "That's not simpler! That's ... that's harder!" From a human perspective looking at the two separate tables and associating the id in the "breed_id" column with the value in the "id" column of the "breeds" table to lookup the name of the breed *is* harder. But, SQL provides tools to make this *very easy*. You will learn about that in the homework, tonight, and in all of the database work that you'll be doing from here on out. Eventually, thinking this way about data will become second nature.

To represent this in SQL, you will need two SQL statements. The first one, the one for the "breeds" table, you should be able to construct that already with the knowledge that you have. It would look like this. Type this into your `plsql` shell.

```
CREATE TABLE breeds (
  id SERIAL,
  name VARCHAR(50) NOT NULL,
  PRIMARY KEY (id)
);
```

Now, here's the new thing. You want the database to make sure that the value in the "breed_id" column of the "puppies" table references the value in the "id" table of the "breeds" table. This reference is called a **foreign key**. That means that the value in the column *must exist* as the value of a primary key in the table that it references. This **referential integrity** is the backbone of relational databases. It prevents bad data from getting put into those foreign key columns.

Here's what the new "puppies" SQL looks like. Drop the old "puppies" table and type this SQL in there.

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

That new thing at the bottom of the definition, that's how you relate one table to another. If follows the syntax

```
FOREIGN KEY («column name in this table»)
  REFERENCES «other table name»(«primary key column in other table»)
```

Looking at the spreadsheets, again, the presence of the foreign key would make it *impossible* for someone to enter a value in the "breed_id" column that did not exist in the "id" column of the "breeds" table.

![Puppies and breed spreadsheets normalized]

You can see that the puppies with ids of 1 and 9, "Cooper" and "Leinni", both have the "breed_id" of 8. That means they're both "Miniature Schnauzers". What if, originally, someone had misspelled "Schnauzers"? If it was still just a text column in the "puppies" sheet, you'd have to go find and replace every single instance of the misspelling. Now, because it's only spelled once and then *referenced*, you would only need to update the misspelling in one place!

Order of table declarations

The order of running these table definitions is important. Because "puppies" now relies on "breeds" to exist for that foreign key relationship, you *must* create the "breeds" table first. If you had tried to create the "puppies" table first, you would see the following error message.

```
ERROR: relation "breeds" does not exist
```

Now that you have both of those tables in your database, what do you think would happen if you tried to drop the "breeds" table? Another table depends on it. When you tried to drop a user that owned a database, you got an error because that database object depended on that user existing, the same things happens now.

Type the SQL to drop the "breeds" table from the database. You should see the following error message.

```
ERROR:  cannot drop table breeds because other objects depend on it
DETAIL:  constraint puppies_breed_id_fkey on table puppies depends on table breeds
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

You can see that PostgreSQL has told you that other things depend on the "breeds" table and, specifically, a thing called "puppies_breed_id_fkey" depends on it. That is the auto-generated name for the foreign key that you created in the "puppies" table. It took the name of the table, the name of the column, and the string "fkey" and joined them all together with underscores.

In the homework for tomorrow, you will see how to *join* together two tables into one virtual table so that the breed names are right there along with the puppies data.

What you've done

In this walk-through, you

- Learned about nullable columns and how to control that behavior by writing `NOT NULL` in your column specifications
- Learned that `NULL` means an "absence of a value" which makes database administrators groan with displeasure
- Learned about how to specify default values for a column
- Learned the purpose of and how to declare integer-valued primary keys for a table using the `PRIMARY KEY` constraint and `SERIAL` data type
- Learned about *normalization* and the steps to refactor a table to remove duplicated data
- Learned the purpose of and how to declare foreign keys to relate the column of one table to the primary key of another table

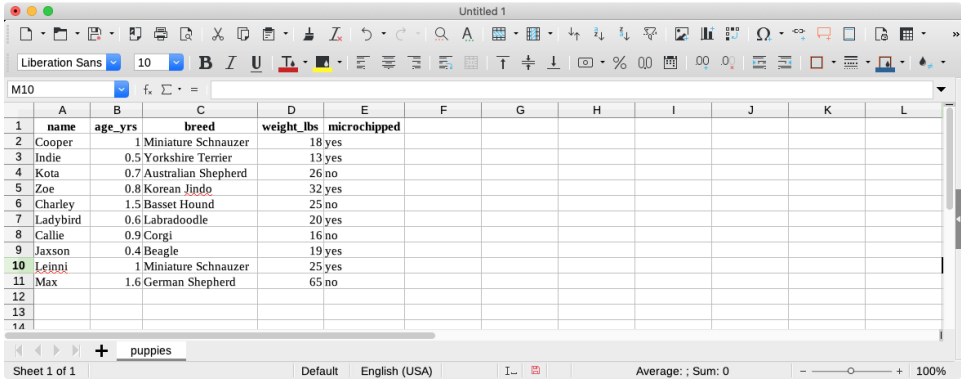
[Puppies and breed spreadsheets normalized]: images/spreadsheet-puppies-and-breeds-normalized.pngimages/spreadsheet-puppies-and-breeds-normalized.pngort "[TOC]" {cmd="toc" depthFrom=2 depthTo=6 orderedList=false} -->

This is a walk-through: Please type along as you read what's going on in this article.

In this walk-through, you will

- Learn about nullable columns,
- Learn about default values for columns,
- Learn how to make columns have unique values,
- Learn about primary keys, and,
- Learn about relating tables through foreign keys to maintain data and referential integrity.

Here is the "puppies" spreadsheet, table definition, and the SQL to create it from the last article.



name	age_yrs	breed	weight_lbs	microchipped
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

Column	JavaScript data type	Max length	ANSI SQL data type
name	string	50	VARCHAR(50)
age_yrs	number		NUMERIC(3,1)
breed	string	100	VARCHAR(100)
weight_lbs	number		INTEGER
microchipped	Boolean		BOOLEAN

```
CREATE TABLE puppies (
  name VARCHAR(50),
  age_yrs NUMERIC(3,1),
  breed VARCHAR(100),
  weight_lbs INTEGER,
  microchipped BOOLEAN
);
```

In this article, you will add more specifications to this table so that you can properly use it. Then, you will refactor it into two tables that relate to one another.

Database Management Project

This project asks you to write some SQL to make some tests pass. You must do them in order of the file name.

You'll be writing your SQL in *.sql files. This is just like writing it in the `psql` client. You put semicolons after each statement. But, you don't have to worry about silly mistakes because it's in a file and they're easy to fix.

Instructions

- Clone the project from https://github.com/appacademy-starters/sql-database-management-starter.
- `cd` into the project folder
- `npm install` to install dependencies in the project root directory
- `npm test` to run the specs
- You can view the test cases in `test`. Your job is to write code in
  - **01-create-users.sql** to write statements that will create users
  - **01-create-databases.sql** to write statements that will create databases
  - **01-create-aa-times-tables.sql** will create related tables in a database

---

WEEK-10 DAY-2
Data! Data! Data!

SQL Learning Objectives

SQL is the language of relational data. It is one of the core languages that most software developers know because of its prevalence in the industry due to the common use of RDBMSes. The objectives for your SQL learning journey cover:

- How to use the `SELECT ... FROM ...` statement to select data from a single table
- How to use the `WHERE` clause on `SELECT`, `UPDATE`, and `DELETE` statements to narrow the scope of the command
- How to use the `JOIN` keyword to join two (or more) tables together into a single virtual table
- How to use the `INSERT` statement to insert data into a table
- How to use an `UPDATE` statement to update data in a table
- How to use a `DELETE` statement to remove data from a table
- How to use a seed file to populate data in a database
- How to perform relational database design
- How to use transactions to group multiple SQL commands into one succeed or fail operation
- How to apply indexes to tables to improve performance
- Explain what and why someone would use `EXPLAIN`

- Demonstrate how to install and use the **node-postgres** library and its `Pool` class to query a PostgreSQL-managed database
- Explain how to write prepared statements with placeholders for parameters of the form "1", "2", and so on

---

Retrieving Rows From A Table Using SELECT

In the first reading, we covered SQL and PostgreSQL and how to set up PostgreSQL. In this reading, we're going to learn how to write a simple SQL query using SELECT.

What is a query?

SQL stands for *Structured Query Language*, and whenever we write SQL we're usually querying a database. A query is simply a question we're asking a database, and we're aiming to get a response back. The response comes back to us as a list of table rows.

Example table

Let's say we had the following database table called `puppies`. We'll use this table to make our queries:

puppies table

name	age_yrs	breed	weight_lbs	microchipped
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

Using psql in the terminal

As we covered in the first reading, psql allows us to access the PostgreSQL server and make queries via the terminal. Open up the terminal on your machine, and connect to the PostgreSQL server by using the following psql command:

```
psql -U postgres
```

The above command lets you access the PostgreSQL server as the user 'postgres' (`-U` stands for user). After you enter this command, you'll be prompted for the password that you set for the 'postgres' user during installation. Type it in, and hit Enter. Once you've successfully logged in, you should see the following in the terminal:

```
Password for user postgres:
psql (11.5, server 11.6)
Type "help" for help.

postgres=#
```

You can exit psql at anytime with the command `\q`, and you can log back in with `psql -U postgres`. (*See this Postgres Cheatsheet for a list of more PSQL commands.*)

We'll use the following PostgreSQL to create the `puppies` table above. After you've logged into the psql server, type the following code and hit Enter.

puppies.sql

```
create table puppies (
  name VARCHAR(100),
  age_yrs NUMERIC(2,1),
  breed VARCHAR(100),
  weight_lbs INT,
  microchipped BOOLEAN
);

insert into puppies
values
('Cooper', 1, 'Miniature Schnauzer', 18, 'yes');

insert into puppies
values
('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');
```

In the above SQL, we created a new table called `puppies`, and we gave it the following columns: `name`, `age_yrs`, `breed`, `weight_lbs`, and `microchipped`. We filled the table with ten rows containing data for each puppy, by using `insert into puppies values ()`.

We used the following PostgreSQL data types: `VARCHAR`, `NUMERIC`, `INT`, and `BOOLEAN`.

- `VARCHAR(n)` is a variable-length character string that lets you store up to *n* characters. Here we've set the character limit to 100 for the `name` and `breed` columns.
- `NUMERIC(p,s)` is a floating-point number with *p* digits and *s* number of places after the decimal point. Here we've set the values for the `age_yrs` column to up to two digits before the decimal and one place after the decimal.
- `INT` is a 4-byte integer, which we've set on the `weight_lbs` column.
- `BOOLEAN` is, of course, a Boolean value. We've set the `microchipped` column to accept Boolean values. SQL accepts the standard Boolean values `true`, `false`, or `null`. However, you'll note that we've used `yes` and `no` in our `microchipped` column because PostgreSQL Booleans can be any of the following values:

TRUE	FALSE
true	false
't'	'f'
'true'	'false'
'yes'	'no'
'y'	'n'
'1'	'0'

Simple SELECT Query

We can write a simple SELECT query to get results back from the table above. The syntax for the SELECT query is `SELECT [columns] FROM [table]`.

SELECT all rows

Using `SELECT *` is a quick way to get back all the rows in a given table. It is discouraged in queries that you write for your applications. Use it only when playing around with data, not for production code.

```
SELECT *
FROM   puppies;
```

Type the query above into your psql terminal, and make sure to add a semicolon at the end, which terminates the statement. `SELECT` and `FROM` should be capitalized. The above query should give us back the entire `puppies` table:

name	age_yrs	breed	weight_lbs	microchipped
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

SELECT by column

We can see all the rows in a given column by using `SELECT [column name]`.

```
SELECT name
FROM   puppies;
```

Type the query above into your psql terminal, and make sure to add a semicolon at the end, which terminates the statement. `SELECT` and `FROM` should be capitalized. The above query should give us back the following:

name
Cooper
Indie
Kota
Zoe
Charley
Ladybird
Callie
Jaxson
Leinni
Max

SELECT multiple columns

To see multiple columns, we can concatenate the column names by using commas between column names.

```
SELECT name
     , age_yrs
     , weight_lbs
FROM   puppies;
```

Type the query above into your psql terminal, and make sure to add a semicolon at the end, which terminates the statement. `SELECT` and `FROM` should be capitalized. The above query should give us back the following:

name	age_yrs	weight_lbs
Cooper	1	18
Indie	0.5	13
Kota	0.7	26
Zoe	0.8	32
Charley	1.5	25
Ladybird	0.6	20
Callie	0.9	16
Jaxson	0.4	19
Leinni	1	25
Max	1.6	65

Formatting SELECT statements

This is another of those hot-button topics with software developers. Some people like to put all the stuff on one line for each SQL keyword.

```
SELECT name, age_yrs, weight_lbs
FROM   puppies;
```

That works for short lists. But some tables have hundreds of columns. That gets long.

Some developers like what you saw earlier, the "each column name on its own line with the comma at the front".

```
SELECT name
     , age_yrs
     , weight_lbs
FROM   puppies;
```

They like this because if they need to comment out a column name, they can just put a couple of dashes at the beginning of the line.

```
SELECT name
--     , age_yrs
     , weight_lbs
FROM   puppies;
```

Some developers just do a word wrap when lines get too long.

All of these are fine. Just stay consistent within a project how you do them.

What we learned:

- What a query is
- How to connect to the PostgreSQL server with psql
- How to construct an example SQL table

- PostgreSQL data types
- How to write a simple SELECT query
- How to SELECT all rows, rows by column, and rows by multiple columns

---

Selecting Table Rows Using WHERE And Common Operators

In the last reading, we learned how to create a simple SQL query using SELECT. In this reading, we'll be adding a WHERE clause to our SELECT statement to further filter a database table and get specific rows back.

Using SELECT and WHERE

Previously, we covered how to use SELECT queries to fetch all of a table's rows or specified table rows by column(s). We can filter information returned by our query by using a WHERE clause in our SELECT statement.

Let's look at some examples of adding a WHERE clause using our `puppies` table from before:

name	age_yrs	breed	weight_lbs	microchipped
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

WHERE clause for a single value

The simplest WHERE clause finds a row by a single column value. See the example below, which finds the rows where the breed equals 'Corgi':

```
SELECT name, breed FROM puppies
  WHERE breed = 'Corgi';
```

`SELECT`, `FROM`, and `WHERE` are capitalized. Notice that the string must use single quotation marks. *Note: PostgreSQL converts all names of tables, columns, functions, etc. to lowercase unless they're double quoted.* For example: `create table Foo()` will create a table called `foo`, and `create table "Bar"()` will create a table called `Bar`. If you use double quotation marks in the query above, you'll get an error that says `column "Corgi" does not exist` because it thinks you're searching for the capitalized column name `Corgi`.

Use the command `psql -U postgres` in the terminal, and type in your 'postgres' user password to connect to the PostgreSQL server. You should have a puppies table in your postgres database from the last reading. Once you're in psql, enter the query above into the terminal and press Enter. You should get back one result for Callie the Corgi.

```
  name  | breed
--------+-------
 Callie | Corgi
```

WHERE clause for a list of values

We can also add a WHERE clause to check for a list of values. The syntax is `WHERE [column] IN ('value1', 'value2', 'value3')`. Let's say we wanted to find the name and breed of the puppies who are Corgis, Beagles, or Yorkshire Terriers. We could do so with the query below:

```
SELECT name, breed FROM puppies
  WHERE breed IN ('Corgi', 'Beagle', 'Yorkshire Terrier');
```

Entering this query into psql should yield the following results:

```
  name  |       breed
--------+-------------------
 Indie  | Yorkshire Terrier
 Callie | Corgi
 Jaxson | Beagle
```

WHERE clause for a range of values

In addition to checking for string values, we can use the WHERE clause to check for a range of numeric/integer values. This time, let's find the name, breed, and age of the puppies who are between 0 to 6 months old.

```
SELECT name, breed, age_yrs FROM puppies
  WHERE age_yrs BETWEEN 0 AND 0.6;
```

Entering this query into psql should yield the following results:

```
   name   |       breed        | age_yrs
----------+--------------------+---------
 Indie    | Yorkshire Terrier  |     0.5
 Ladybird | Labradoodle        |     0.6
 Jaxson   | Beagle             |     0.4
```

ORDER BY

Getting the values back from a database in any order it wants to give them to you is ludicrous. Instead, you will often want to specify the order in which you get them back. Say you wanted them in alphabetical order by their name. Then, you would write

```
SELECT name, breed
FROM puppies
ORDER BY name;
```

Say you wanted that returned from oldest dog to youngest dog. You would write

```
SELECT name, breed
FROM puppies
ORDER BY age_yrs DESC;
```

where `DESC` means in descending order. Note that the column that you order on does not have to appear in the column list of the `SELECT` statement.

LIMIT and OFFSET

Say your query would return one million rows because you've cataloged every puppy in the world. That would be a lot for any application to handle. Instead, you may want to limit the number of rows returned. You can do that with the `LIMIT` keyword.

```
SELECT name, breed
FROM puppies
ORDER BY age_yrs
LIMIT 100;
```

That would return the name and breed of the 100 youngest puppies. (Why?) That is, of the million rows that the statement would find, it *limits* the number to only 100.

Let's say you want to see the *next* 100 puppies after the first hundred. You can do that with the `OFFSET` keyword which comes after the `LIMIT` clause.

```
SELECT name, breed
FROM puppies
ORDER BY age_yrs
LIMIT 100 OFFSET 100;
```

That will return only rows 101 - 200 of the result set. It *limits* the total number of records to return to 100. Then, it starts at the 100th row and counts 100 records. Those are the records returned.

SQL operators

A SQL operator is a word or character that is used inside a WHERE clause to perform comparisons or arithmetic operations. In the three examples above, we used SQL operators inside of WHERE clauses to filter table rows -- `=`, `IN`, `BETWEEN`, and `AND`.

The following is a listing of SQL operators. We can combine any of these operators in our query or use a single operator by itself.

Logical operators

Operator	Description
ALL	TRUE if all of the subquery values meet the condition.
AND	TRUE if all the conditions separated by AND are TRUE.
ANY	TRUE if any of the subquery values meet the condition.
BETWEEN	TRUE if the operand is within the range of comparisons.
EXISTS	TRUE if the subquery returns one or more records.
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern (accepts "wildcards").
NOT	Displays a record if the condition(s) is NOT TRUE.
OR	TRUE if any of the conditions separated by OR is TRUE.
SOME	TRUE if any of the subquery values meet the condition.

Here is another example query with a WHERE clause using several logical operators: `NOT`, `IN`, `AND`, and `LIKE`.

```
SELECT name, breed FROM puppies
  WHERE breed NOT IN ('Miniature Schnauzer', 'Basset Hound', 'Labradoodle')
    AND breed NOT LIKE '%Shepherd';
```

Note: Pay attention to that `LIKE` operator. You will use it more than you want to. The wildcard it uses is the percent sign. Here's a table to help you understand.

`LIKE`	Matches "dog"	Matches "hotdog"	Matches "dog-tired"	Matches "ordogordo"
`'dog'`	yes	no	no	no

`LIKE`	Matches "dog"	Matches "hotdog"	Matches "dog-tired"	Matches "ordogordo"
`'%dog'`	yes	yes	no	no
`'dog%'`	yes	no	yes	no
`'%dog%'`	yes	yes	yes	yes

Entering this query into psql should yield the following results:

```
  name  |       breed
--------+-------------------
 Indie  | Yorkshire Terrier
 Zoe    | Korean Jindo
 Callie | Corgi
 Jaxson | Beagle
```

With the query above, we filtered out six puppies: two Miniature Schnauzers, one Basset Hound, one Labradoodle, and two Shepherds. We started with ten puppies in the table, so we're left with four table rows. There are two puppies who are Shepherd breeds: an Australian Shepherd and a German Shepherd. We used the `LIKE` operator to filter these. In `'%Shepherd`, the `%` matches any substring value before the substring 'Shepherd'.

Arithmetic operators

Operator	Meaning	Syntax
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (returns remainder)	a % b

Here is an example query with a WHERE clause using the multiplication operator to find puppies that are 6 months old:

```
SELECT name, breed, age_yrs FROM puppies
  WHERE age_yrs * 10 = 6;
```

Entering the above query into psql will yield one result:

```
   name   |    breed    | age_yrs
----------+-------------+---------
 Ladybird | Labradoodle |     0.6
```

Comparison operators

Operator	Meaning	Syntax
=	Equals	a = b
!=	Not equal to	a != b
<>	Not equal to	a <> b
Greater than		a > b
<	Less than	a < b
>=	Greater than or equal to	a >= b
<=	Less than or equal to	a <= b
!<	Not less than	a !< b
!>	Not greater than	a !> b

Here is an example query with a WHERE clause using the `>` comparison operator:

```
SELECT name, breed, weight_lbs FROM puppies
  WHERE weight_lbs > 50;
```

Entering the above query into psql will yield one result:

```
 name |      breed      | weight_lbs
------+-----------------+------------
 Max  | German Shepherd |         65
```

What we learned:

- How to use a WHERE clause in a SELECT query
- How to construct WHERE clauses matching a single value, a list of values, and a range of values
- What SQL operators are
- How to use logical operators, arithmetic operators, and comparison operators

---

Inserting Data Into A Table

If you have data, but it's not in tables, does the data even exist? Not to an app! We often need to create relational databases on the back end of the Web apps we're building so that we can ultimately display this data on the front end of our application. All relational database data is stored in tables, so it's important to learn how to create tables and successfully query them.

Of the four data manipulation statements, `INSERT` is the easiest.

Create a new database named "folks". Now, create a new table named "friends" with the following column specifications.

Name	Data type	Constraints
id	`SERIAL`	`PRIMARY KEY`
first_name	`VARCHAR(255)`	`NOT NULL`
last_name	`VARCHAR(255)`	`NOT NULL`

Now that we have a new table, we need to add table rows with some data. We can insert a new table row using the following syntax:

```
INSERT INTO table_name
VALUES
  (column1_value, colum2_value, column3_value);
```

Let's fill out our "friends" table with information about five friends. In psql, enter the following to add new table rows. *Note the use of single quotation marks for string values. Also note that, since we used the* `SERIAL` *pseudo-type to auto-increment the ID values, we can simply write* `DEFAULT` *for the ID values when inserting new table rows.*

```
INSERT INTO friends (id, first_name, last_name)
VALUES
  (DEFAULT, 'Amy', 'Pond');
```

You can also completely omit the `DEFAULT` keyword if you specify the names of the columns that you want to insert into.

You can also use the "multiple values" insert. This prevents you from having to write `INSERT` with every statement. Even better, if one fails, they all fail. That can help protect your data integrity.

```
INSERT INTO friends (first_name, last_name)
VALUES
('Rose', 'Tyler'),
('Martha', 'Jones'),
('Donna', 'Noble'),
('River', 'Song');
```

Use `SELECT * FROM friends;` to verify that there are rows in the “friends” table:

```
appacademy=# SELECT * FROM friends;
 id | first_name | last_name
----+------------+-----------
  1 | Amy        | Pond
  2 | Rose       | Tyler
  3 | Martha     | Jones
  4 | Donna      | Noble
  5 | River      | Song
```

Now let’s try to insert a new row using the ID of 5:

```
INSERT INTO friends (id, first_name, last_name)
VALUES (5, 'Jenny', 'Who');
```

Because ID is a primary key and that ID is already taken, we should get a message in psql that it already exists:

```
appacademy=# insert into friends values (5, 'Jenny', 'Who');
ERROR:  duplicate key value violates unique constraint "friends_pkey"
DETAIL:  Key (id)=(5) already exists.
```

What we learned:

- How to connect to an existing PostgreSQL database
- How to create a new PostgreSQL database
- How to create a new database table
- Accepted PostgreSQL data types
- How to add new rows to a database table
- What a primary key is/does

Foreign Keys And The JOIN Operation

In relational databases, *relationships* are key. We can create relationships, or *associations*, among tables so that they can access and share data. In a SQL database, we create table associations through *foreign keys* and *primary keys*.

You've learned about primary and foreign keys. Now, it's time to put them to use.

Setting up the database

Create a new database called "learn_joins". Connect to that database. Run the following SQL statements to create tables and the data in them.

```
CREATE TABLE breeds (
  id SERIAL,
  name VARCHAR(50) NOT NULL,
  PRIMARY KEY (id)
);
```

```
INSERT INTO breeds (name)
VALUES
('Australian Shepherd'),
('Basset Hound'),
('Beagle'),
('Corgi'),
('German Shepherd'),
('Korean Jindo'),
('Labradoodle'),
('Miniature Schnauzer'),
('Yorkshire Terrier');
```

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

```
INSERT INTO puppies (name, age_yrs, breed_id, weight_lbs, microchipped)
VALUES
('Cooper', 1, 8, 18, true),
('Indie', 0.5, 9, 13, true),
('Kota', 0.7, 1, 26, false),
('Zoe', 0.8, 6, 32, true),
('Charley', 1.5, 2, 25, false),
('Ladybird', 0.6, 7, 20, true),
('Callie', 0.9, 4, 16, false),
('Jaxson', 0.4, 3, 19, true),
('Leinni', 1, 8, 25, true),
('Max', 1.6, 5, 65, false);
```

Using JOIN to retrieve rows from multiple tables

Now that we've set up an association between the "puppies" table and the "friends" table, we can access data from both tables. We can do so by using a JOIN operation in our SELECT query. Type the following into psql:

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

The `JOIN` operation above is joining the "puppies" table with the "breeds" table together into a single table using the foreign key/primary key shared between the two tables: `breed_id`.

You should get all rows back containing all information for the puppies and breeds with matching `breed_id` values:

```
postgres=# SELECT * FROM puppies
postgres-# INNER JOIN breeds ON (puppies.breed_id = breeds.id);
 id |   name   | age_yrs | breed_id | weight_lbs | microchipped | id |        nam
----+----------+---------+----------+------------+--------------+----+-----------
  1 | Cooper   |     1.0 |        8 |         18 | t            |  8 | Miniature
  2 | Indie    |     0.5 |        9 |         13 | t            |  9 | Yorkshire
  3 | Kota     |     0.7 |        1 |         26 | f            |  1 | Australian
  4 | Zoe      |     0.8 |        6 |         32 | t            |  6 | Korean Jin
  5 | Charley  |     1.5 |        2 |         25 | f            |  2 | Basset Hou
  6 | Ladybird |     0.6 |        7 |         20 | t            |  7 | Labradoodl
  7 | Callie   |     0.9 |        4 |         16 | f            |  4 | Corgi
  8 | Jaxson   |     0.4 |        3 |         19 | t            |  3 | Beagle
  9 | Leinni   |     1.0 |        8 |         25 | t            |  8 | Miniature
 10 | Max      |     1.6 |        5 |         65 | f            |  5 | German She
(10 rows)
```

We could make our query more specific by selecting specific columns, adding a `WHERE` clause, or doing any number of operations that we could do in a normal

`SELECT` query. Aside from an `INNER JOIN`, we could also do different types of `JOIN` operations. (Refer to this overview on PostgreSQL JOINS for more information.)

What we learned:

- What a foreign key is/does
- How to create a foreign key
- How to alter/update an existing table
- How to use the JOIN operation to get rows from two tables

Helpful links:

- PostgreSQL Docs: Constraints
- PostgreSQL Docs: Data Types > Numeric Types
- PostgreSQL Docs: Joins Between Tables
- PostgreSQL Tutorial: PostgreSQL Joins

Writing And Running A Seed File In PSQL

After a database is created, we need to populate it, or *seed* it, with data. Until now, we've used the command-line psql interface to create tables and insert rows into those tables. While that's fine for small datasets, it would be cumbersome to add a large dataset using the command line.

In this reading, we'll learn how to create and run a seed file, which makes the process of populating a database with test data much easier.

Creating a seed file

You can create a seed file by opening up VSCode or any text editor and saving a file with the `.sql` extension.

Let's create a seed file called `seed-data.sql` that's going to create a new table called `pies` and insert 50 pie rows into the table. Use the code below to create the seed file, and make sure to save your seed file on your machine.

seed-data.sql

```
CREATE TABLE pies (
  flavor VARCHAR(255) PRIMARY KEY,
  price FLOAT
);

INSERT INTO pies VALUES('Apple', 19.95);
INSERT INTO pies VALUES('Caramel Apple Crumble', 20.53);
INSERT INTO pies VALUES('Blueberry', 19.31);
INSERT INTO pies VALUES('Blackberry', 22.86);
INSERT INTO pies VALUES('Cherry', 22.32);
INSERT INTO pies VALUES('Peach', 20.45);
INSERT INTO pies VALUES('Raspberry', 20.99);
INSERT INTO pies VALUES('Mixed Berry', 21.45);
INSERT INTO pies VALUES('Strawberry Rhubarb', 24.81);
INSERT INTO pies VALUES('Banana Cream', 18.66);
INSERT INTO pies VALUES('Boston Toffee', 25.00);
INSERT INTO pies VALUES('Banana Nutella', 22.12);
INSERT INTO pies VALUES('Bananas Foster', 24.99);
INSERT INTO pies VALUES('Boston Cream', 23.75);
INSERT INTO pies VALUES('Cookies and Cream', 18.27);
INSERT INTO pies VALUES('Coconut Cream', 22.89);
INSERT INTO pies VALUES('Chess', 25.00);
INSERT INTO pies VALUES('Lemon Chess', 25.00);
INSERT INTO pies VALUES('Key Lime', 19.34);
INSERT INTO pies VALUES('Lemon Meringue', 19.58);
INSERT INTO pies VALUES('Guava', 18.92);
INSERT INTO pies VALUES('Mango', 19.55);
INSERT INTO pies VALUES('Plum', 20.21);
INSERT INTO pies VALUES('Apricot', 20.55);
INSERT INTO pies VALUES('Gooseberry', 23.54);
INSERT INTO pies VALUES('Lingonberry', 24.35);
INSERT INTO pies VALUES('Pear Vanilla Butterscotch', 25.13);
INSERT INTO pies VALUES('Baked Alaska', 25.71);
INSERT INTO pies VALUES('Chocolate', 19.00);
INSERT INTO pies VALUES('Chocolate Mousse', 21.46);
INSERT INTO pies VALUES('Mexican Chocolate', 28.33);
INSERT INTO pies VALUES('Chocolate Caramel', 26.67);
INSERT INTO pies VALUES('Chocolate Chip Cookie Dough', 18.65);
INSERT INTO pies VALUES('Pecan', 26.33);
INSERT INTO pies VALUES('Bourbon Caramel Pecan', 27.88);
INSERT INTO pies VALUES('Chocolate Pecan', 27.63);
INSERT INTO pies VALUES('Pumpkin', 20.91);
INSERT INTO pies VALUES('Sweet Potato', 20.75);
INSERT INTO pies VALUES('Purple Sweet Potato', 26.34);
INSERT INTO pies VALUES('Cheesecake', 28.99);
INSERT INTO pies VALUES('Butterscotch Praline', 29.78);
INSERT INTO pies VALUES('Salted Caramel', 30.32);
INSERT INTO pies VALUES('Salted Honey', 59.00);
INSERT INTO pies VALUES('Sugar Cream', 33.89);
INSERT INTO pies VALUES('Mississippi Mud', 28.67);
INSERT INTO pies VALUES('Turtle Fudge', 30.58);
INSERT INTO pies VALUES('Fruit Loops', 20.45);
INSERT INTO pies VALUES('Black Forest', 34.99);
INSERT INTO pies VALUES('Maple Cream', 35.88);
INSERT INTO pies VALUES('Smores', 26.43);
INSERT INTO pies VALUES('Milk Bar', 46.00);

SELECT * FROM pies;
```

Populating a database via < (“left caret”)

Now that you have a seed file, you can insert it into a database with a simple command.

Create a database named "bakery".

The syntax is `psql -d [database] < [path_to_file/file.sql]`. The left caret (`<`) takes the standard input from the file on the right (your seed file) and inputs it into the program on the left (`psql`).

Open up your terminal, and enter the following command. Make sure to replace `path_to_my_file` with the actual file path.

```
psql -d bakery < path_to_my_file/seed-data.sql
```

In the terminal, you should see a bunch of `INSERT` statements and the entire “pies” table printed out (from the `SELECT *` query in the seed file).

You can log into psql and use `\dt` to verify that your new table has been added to the database:

```
postgres=# \dt
List of relations
 Schema |     Name      | Type  |  Owner
 public | breeds        | table | appacademy
 public | pies          | table | appacademy
 public | puppies       | table | appacademy
```

Populating the database via | (“pipe”)

You could also use the “pipe” (`|`) to populate the database with your seed file.

The syntax is `cat [path_to_file/file.sql] | psql -d [database]`. ‘cat’ is a standard Unix utility that reads files sequentially, writing them to standard output. The “pipe” (`|`) takes the standard output of the command on the left and pipes it as standard input to the command on the right.

Try out this method in your terminal. If you have an existing “pies” table, you’ll need to drop this table before you can add it again:

```
DROP TABLE pies;
```

Then, enter the following. Make sure to replace `path_to_my_file` with the actual file path.

```
cat path_to_my_file/seed-data.sql | psql -d postgres
```

Again, you should see a bunch of `INSERT` statements and the entire “pies” table printed out (from the `SELECT *` query in the seed file).

You can log into psql and use `\dt` to verify that your new table has been added to the database:

```
postgres=# \dt
List of relations
 Schema |     Name      | Type  |  Owner
 public | friends       | table | postgres
 public | pies          | table | postgres
 public | puppies       | table | postgres
```

What we learned:

- What a seed file is
- How to create a seed file
- How to populate a database with a seed file using <
- How to populate a database with a seed file using |

Create And Seed A Database Project

In our SQL readings, we installed PostgreSQL, and we learned how to create a new PostgreSQL database and how to create and run a seed file in our database.

In this project, you'll practice seeding your database with a seed file that's full of, well, *seeds*! Hopefully when you're done, you'll have *grown* your SQL knowledge.

Project overview

Practice creating a new database and piping a seed file into your database.

- In Phase 1, you'll create a new database.
- In Phase 2, you'll create a new seed data file.
- In Phase 3, you'll pipe your seed data in your database via `psql` on the command line.

Phase 1: Create a new database

First, log into `psql` on the command line as your user.

Second, create a new database in PostgreSQL called `farm` by using the following syntax:

```
CREATE DATABASE [databasename];
```

Note: You can check to make sure you've created a new database by viewing the list of databases with `\l`.

Phase 2: Create a seeds seed file

Create a seed file full of seeds! Set up a SQL file with seed data that will produce two tables: an "edible_seeds" with 40 rows and a "flower_seeds" table with 20 rows.

Edible Seeds

id	name	type	price_per_pound	in_stock
1	Chia	Pseudocereal	6.95	yes
2	Flax	Pseudocereal	6.90	yes
3	Amaranth	Pseudocereal	14.99	yes
4	Quinoa	Pseudocereal	12.49	no
5	Sesame	Pseudocereal	13.49	yes
6	Hemp	Other	10.99	yes
7	Chickpea	Legume	7.99	yes
8	Pea	Legume	7.50	no
9	Soybean	Legume	12.99	yes
10	Acorn	Nut	11.99	yes
11	Almond	Nut	13.99	yes
12	Beech	Nut	10.94	yes
13	Chestnut	Nut	13.99	yes
14	Water Chestnut	Nut	9.99	no
15	Macadamia	Nut	25.00	yes
16	Pistachio	Nut	20.00	yes
17	Pine nuts	Nut-like gymnosperm	23.00	yes
18	Pecan	Nut	6.99	yes
19	Juniper berries	Nut-like gymnosperm	11.99	yes
20	Cashew	Nut	23.61	yes
21	Hazelnut	Nut	25.49	yes

id	name	type	price_per_pound	in_stock
22	Sunflower seed	Other	9.99	yes
23	Poppy seed	Other	12.99	yes
24	Barley	Cereal	9.99	yes
25	Maize	Cereal	6.98	yes
26	Oats	Cereal	9.99	yes
27	Rice	Cereal	7.90	yes
28	Rye	Cereal	9.87	yes
29	Spelt	Cereal	7.50	yes
30	Wheat berries	Cereal	2.50	no
31	Buckwheat	Pseudocereal	5.60	yes
32	Jackfruit	Other	15.00	yes
33	Durian	Other	8.26	no
34	Lotus seed	Other	12.99	yes
35	Ginko	Nut-like gymnosperm	12.80	yes
36	Peanut	Legume	5.99	yes
37	Pumpkin seed	Other	5.99	yes
38	Watermelon seed	Other	6.99	yes
39	Pomegranate seed	Other	27.63	yes
40	Cacao bean	Other	12.99	yes

Use the following data types for your "edible_seeds" columns:

- id - SERIAL
- name - VARCHAR that accepts 255 characters
- type - VARCHAR that accepts 255 characters
- price_per_pound - FLOAT or REAL
- in_stock - BOOLEAN

Flower Seeds

id	name	main_color	seeds_per_packet	price_per_packet
1	Begonia Fiona Red	Red	25	4.95
2	Moonflower Seeds	White	25	2.95
3	Easy Wave F1 Lavender Sky Blue Petunia Seeds	Lavender	10	4.25
4	Super Hero Spry Marigold Seeds	Marigold	50	2.95
5	Zinnia Zinderella Lilac	Pink	25	3.95
6	Mini Ornamental Mint Seeds	Green	10	3.95

id	name	main_color	seeds_per_packet	price_per_packet
7	Kabloom Light Pink Blast Calibrachoa	Green	10	4.95
8	Calibrachoa Kabloom Coral	Coral	10	4.95
9	Fiesta del Sol Mexican Sunflower Seeds	Red	30	3.95
10	Cosmos Apricot Lemonade	Yellow	25	3.95
11	Zinderella Purple Zinnia Seeds	Purple	25	3.95
12	Fireball Marigold Seeds	Varies	25	3.95
13	Gerbera Revolution Bicolor Red Lemon	Red	10	8.95
14	Paradise Island Calibrachoa Fuseables Seeds	Varies	5	6.95
15	Cheyenne Spirit Coneflower Seeds	Varies	15	7.95
16	Leucanthemum Madonna	White	25	4.95
17	Zinnia Zinderella Peach	Peach	25	3.95
18	Kabloom Orange Calibrachoa	Orange	10	4.95
19	Fountain Blue Lobelia Seeds	Blue	100	2.50
20	Envy Zinnia Seeds	Green	50	2.95

Use the following data types for your "flower_seeds" columns:

- id - SERIAL
- name - VARCHAR that accepts 300 characters
- main_color - VARCHAR that accepts 100 characters
- seeds_per_packet - INT
- price_per_packet - FLOAT
- in_stock - BOOLEAN

Note: Make sure to save your seed file on your machine so that you can pipe it into your database in the next phase!

Phase 3: Pipe your seed file into your new database

After you've saved your seed file, use the caret and pipe methods to seed your `farm` database with the data from the "edible_seeds" table. (*Note: Make sure you've quit `psql` first with `\q`.*)

There are two ways to seed your database:

Method 1. Seed your database via caret method

```
psql -d [database] -U [username] < [path_to_file/file.sql]
```

Method 2. Seed your database via pipe method

```
cat [path_to_file/file.sql] | psql -d [database] -U [username]
```

Try both of these methods.

If you want to seed using the file again, you need to first drop the tables.

Access the database by:

```
psql -d [database] -U [username]
```

Then drop the tables:

```
DROP TABLE [table];
```

Then you can run the seed file again using any of the two above methods.

Check to make sure your seed file has actually updated the `farm` database

- Log into `psql` again.
- Connect to the `farm` database (syntax: `\c [database]`).
- Make sure your "edible_seeds" table is filled with seeds: `SELECT * FROM «table name»;`
- Make sure your "flower_seeds" table is filled with seeds: `SELECT * FROM «table name»;`

Solving The SQL Menagerie Project

In our SQL readings, we learned how to write basic SQL queries and incorporate `WHERE` clauses to filter for more specific results. We also learned how to use a `JOIN` operation to get information from multiple tables.

In this project, put your SQL knowledge to the test and show off your querying skills.

We've put together a collection (let's call it a *menagerie*) of SQL problems for you to solve below. Solve them all, and you'll be the master of the menagerie!

Getting started

Clone the starter repository from https://github.com/appacademy-starters/sql-select-exercises-starter.

Project overview

1. In Phase 1, pipe the seed file into a new database.
2. In Phase 2, query the seed tables with basic `SELECT` statements.
3. In Phase 3, query the seed tables using `WHERE` clauses to get more specific rows back.
4. In Phase 4, use a `JOIN` operation to get data from both seed tables.
5. Bonuses! Go beyond what we learned in the readings to deepen your SQL query knowledge.

Phase 1: Pipe in a seed file to create new database tables

We've set up a seed file for you to use in this project called ***cities_and_airports.sql*** that will create two tables: a "cities" table and an "airports" table. These tables show the top 25 most populous U.S. cities and their airports, respectively. Pipe this file into your database, and use these tables for the rest of the project phases.

Go through the following steps:

1. Log into `psql`.
2. Create a new database called "travel".
3. Pipe the *cities_and_airports.sql* seed file into the "travel" database.
4. Check that there's data in both the "cities" and "airports" tables.

Phase 2: Write basic SELECT statements

Retrieve rows from a table using `SELECT` FROM SQL statements.

1. Write a SQL query that returns the city, state, and estimated population in 2018 from the "cities" table.
2. Write a SQL query that returns all of the airport names contained in the "airports" table.

Phase 3: Add WHERE clauses

Select specific rows from a table using WHERE and common operators.

1. Write a SQL query that uses a `WHERE` clause to get the estimated population in 2018 of the city of San Diego.
2. Write a SQL query that uses a `WHERE` clause to get the city, state, and estimated population in 2018 of cities in this list: Phoenix, Jacksonville, Charlotte, Nashville.
3. Write a SQL query that uses a `WHERE` clause to get the cities with an estimated 2018 population between 800,000 and 900,000 people. Show the city, state, and estimated population in 2018 columns.
4. Write a SQL query that uses a `WHERE` clause to get the names of the cities that had an estimated population in 2018 of at least 1 million people (or 1,000,000 people).
5. Write a SQL query to get the city and estimated population in 2018 in number of millions (*i.e. without zeroes at the end: 1 million*), and that uses a `WHERE` clause to return only the cities in Texas.
6. Write a SQL query that uses a `WHERE` clause to get the city, state, and estimated population in 2018 of cities that are NOT in the following states: New York, California, Texas.
7. Write a SQL query that uses a `WHERE` clause with the `LIKE` operator to get the city, state, and estimated population in 2018 of cities that start with the letter "S". (*Note: See the PostgreSQL doc on Pattern Matching for more information.*)
8. Write a SQL query that uses a `WHERE` clause to find the cities with either a land area of over 400 square miles OR a population over 2 million people (or 2,000,000 people). Show the city name, the land area, and the estimated population in 2018.
9. Write a SQL query that uses a `WHERE` clause to find the cities with either a land area of over 400 square miles OR a population over 2 million people

(or 2,000,000 people) -- but not the cities that have both. Show the city name, the land area, and the estimated population in 2018.

10. Write a SQL query that uses a `WHERE` clause to find the cities where the population has increased by over 200,000 people from 2010 to 2018. Show the city name, the estimated population in 2018, and the census population in 2010.

Phase 4: Use a JOIN operation

Retrieve rows from multiple tables joining on a foreign key.

The "airports" table has a foreign key called `city_id` that references the `id` column in the "cities" table.

1. Write a SQL query using an INNER JOIN to join data from the "cities" table with data from the "airports" table using the `city_id` foreign key. Show the airport names and city names only.
2. Write a SQL query using an INNER JOIN to join data from the "cities" table with data from the "airports" table to find out how many airports are in New York City using the city name. (*Note: Use the aggregate function COUNT() to count the number of matching rows.*)

Bonuses

1. **Apostrophe:** Write a SQL query to get all three ID codes (*the Federal Aviation Administration (FAA) ID, the International Air Transport Association (IATA) ID, and the International Civil Aviation Organization (ICAO) ID*) from the "airports" table for Chicago O'Hare International Airport. (*Note: You'll need to escape the quotation mark in O'Hare. See How to include a single quote in a SQL query.*)
2. **Formatting Commas:** Refactor Phase 2, Query #1 to turn the INT for estimated population in 2018 into a character string with commas. (*Note: See Data Type Formatting Functions*)
    - Phase 2, Query #1: Write a SQL query that returns the city, state, and estimated population in 2018 from the "cities" table.
3. **Decimals and Rounding:** Refactor Phase 3, Query #5 to turn number of millions from an integer into a decimal rounded to a precision of two decimal places. (*Note: See Numeric Types and the ROUND function.*)
    - Phase 3, Query #5: Write a SQL query to get the city and estimated population in 2018 in number of millions (*i.e. without zeroes at the end: 1 million*), and that uses a `WHERE` clause to return only the cities in Texas.
4. **ORDER BY and LIMIT Clauses:** Refactor Phase 3, Query #10 to return only one city with the biggest population increase from 2010 to 2018. Show the city name, the estimated population in 2018, and the census population in 2010 for that city. (_Note: You'll do the same calculation as before, but instead of comparing it to 200,000, use the ORDER BY Clause with the LIMIT Clause to sort the results and grab only the top result.)
    - Phase 3, Query #10: Write a SQL query that uses a `WHERE` clause to find the cities where the population has increased by over 200,000 people from 2010 to 2018. Show the city name, the estimated population in 2018, and the census population in 2010.

WEEK-10 DAY-3
Deeper Into Data

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

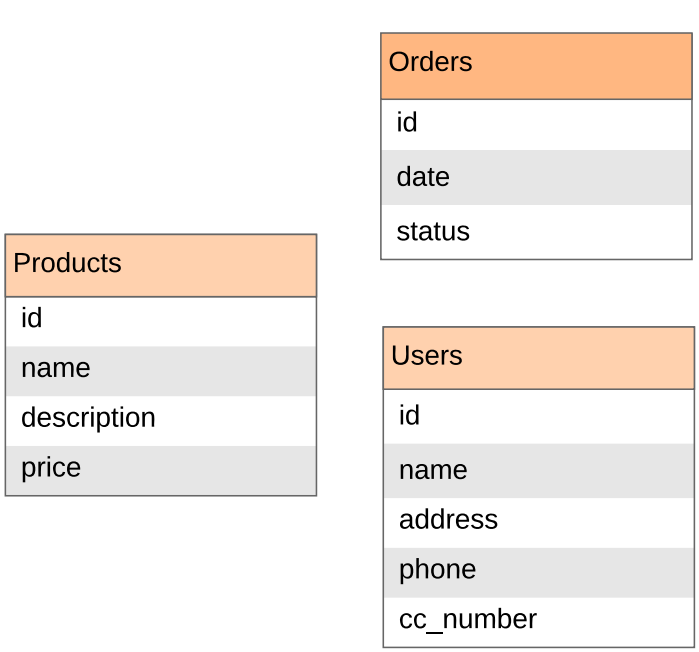
There are four generally-agreed-upon stages of Relational Database Design:

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

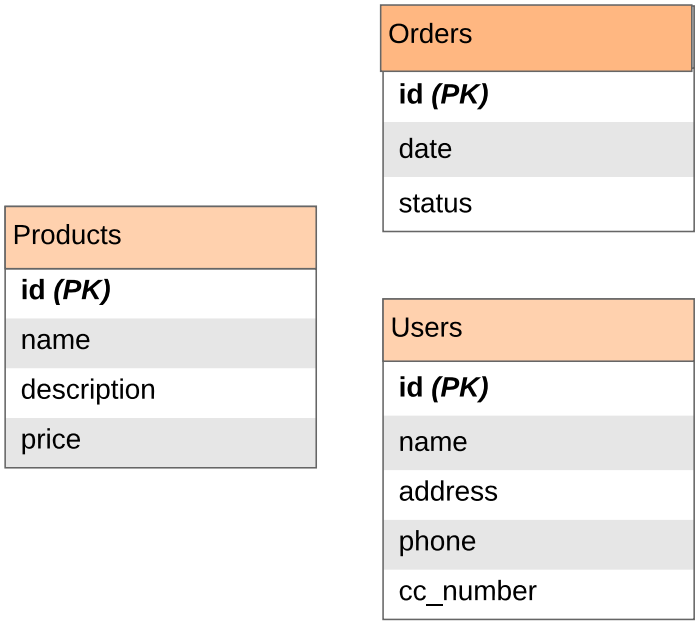
The first stage is identifying the purpose of the database (*Why is the database being created? What problem is it solving? What is the data used for?*), as well as identifying the main entities, or *tables*, that need to be created. It also typically involves identifying the table's attributes (i.e.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

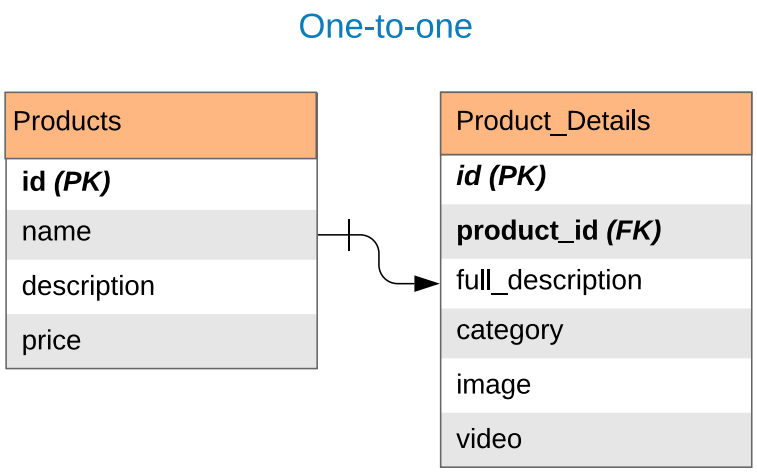
The third stage is to establish the relationships among the tables in the database. There are three types of relational database table relationships:

- 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

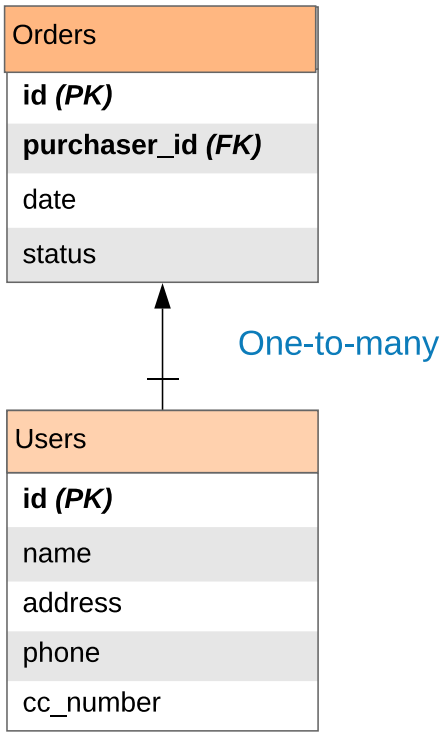
Product Details

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.
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

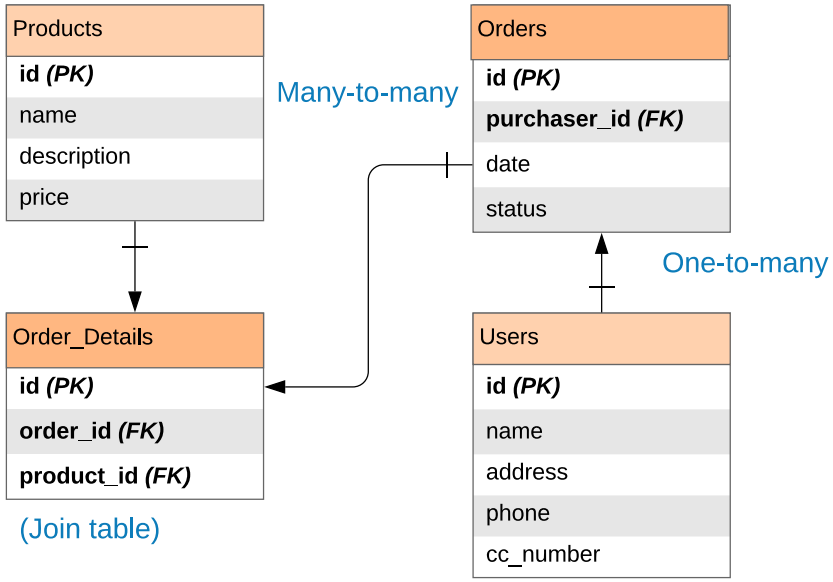
Orders

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

id	order_id	product_id
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?

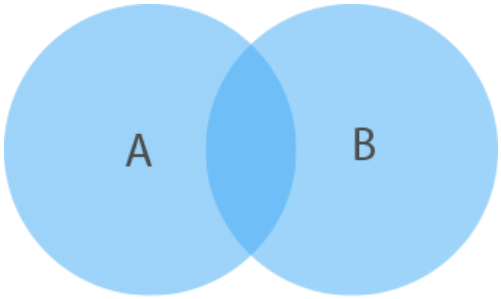
We briefly looked at the `JOIN` operation after we created foreign keys in a previous reading. The JOIN operation allows us to retrieve rows from multiple tables.

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 |        nam
----+-----------+---------+----------+------------+--------------+----+-----------
  1 | Cooper    |     1.0 |        8 |         18 | t            |  8 | Miniature
  2 | Indie     |     0.5 |        9 |         13 | t            |  9 | Yorkshire
  3 | Kota      |     0.7 |        1 |         26 | f            |  1 | Australian
  4 | Zoe       |     0.8 |        6 |         32 | t            |  6 | Korean Jin
  5 | Charley   |     1.5 |        2 |         25 | f            |  2 | Basset Hou
  6 | Ladybird  |     0.6 |        7 |         20 | t            |  7 | Labradoodl
  7 | Callie    |     0.9 |        4 |         16 | f            |  4 | Corgi
  8 | Jaxson    |     0.4 |        3 |         19 | t            |  3 | Beagle
  9 | Leinni    |     1.0 |        8 |         25 | t            |  8 | Miniature
 10 | Max       |     1.6 |        5 |         65 | f            |  5 | German She
```

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  | weight_lbs |
----+------------+-----------+----------+--------+---------+--------+------------
  5 | River      | Song      |        8 | Jaxson |     0.4 | Beagle |         19
(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_code);
```

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 (city_id = 2
```

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..12.700 ro
  ->  Bitmap Heap Scan on tenk1 t1  (cost=2.37..232.35 rows=106 width=244) (actua
        Recheck Cond: (unique1 < 100)
        ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..2.37 rows=106 width=0
              Index Cond: (unique1 < 100)
  ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..3.01 rows=1 width=2
        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 "$1" placeholders is a positional argument for the parameters that you pass in. The "$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".

1".*The'age'variableisinthesecondpositionofthearray,soitwillbesubstitu*
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: 18, microc
  { id: 2, name: 'Indie', age_yrs: '0.5', breed: 'Yorkshire Terrier', weight_lbs: 13, microchip
  { id: 3, name: 'Kota', age_yrs: '0.7', breed: 'Australian Shepherd', weight_lbs: 26, microchi
  { id: 4, name: 'Zoe', age_yrs: '0.8', breed: 'Korean Jindo', weight_lbs: 32, microchipped: tr
  { id: 5, name: 'Charley', age_yrs: '1.5', breed: 'Basset Hound', weight_lbs: 25, microchipped
  { id: 6, name: 'Ladybird', age_yrs: '0.6', breed: 'Labradoodle', weight_lbs: 20, microchipped
  { id: 7, name: 'Callie', age_yrs: '0.9', breed: 'Corgi', weight_lbs: 16, microchipped: false
  { id: 8, name: 'Jaxson', age_yrs: '0.4', breed: 'Beagle', weight_lbs: 19, microchipped: true
  { id: 9, name: 'Leinni', age_yrs: '1.0', breed: 'Miniature Schnauzer', weight_lbs: 25, microc
  { id: 10, name: 'Max', age_yrs: '1.6', breed: 'German Shepherd', weight_lbs: 65, microchipped
```

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.

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.

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:

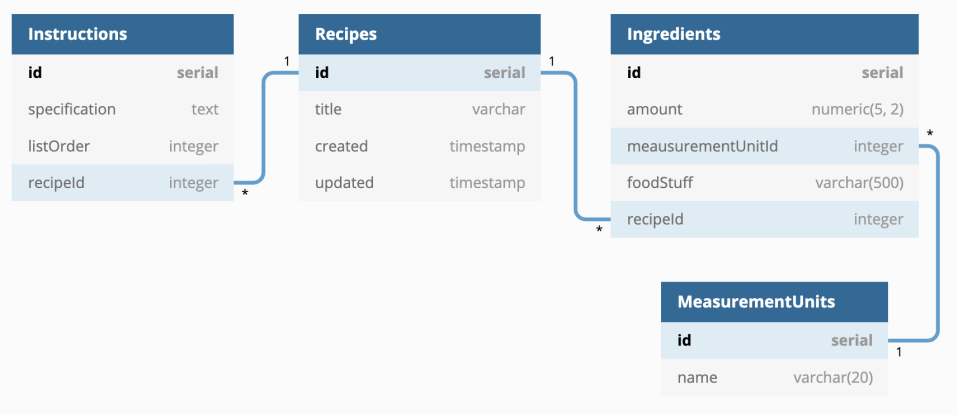
- An amount (optional),
- A unit of measure (optional),
- The actual food stuff, and
- The id of the recipe that it belongs to.

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.

WEEK-10 DAY-4
Sequelize ORM

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

Installing And Using Sequelize

Now that you have gained experience with SQL, it is time to learn how to access data stored in a SQL database using a JavaScript program. You will use a JavaScript library called Sequelize to do this. Sequelize is an example of an *Object Relational Mapping* (commonly abbreviated *ORM*). An ORM allows a JavaScript programmer to fetch and store data in a SQL database using JavaScript functions instead of writing SQL code.

When you finish this reading you will be able to:

- Describe what an Object Relational Mapping is and what it is used for.
- Install and configure the packages needed to use Sequelize.
- Use Sequelize to generate JavaScript code that fetches and stores data in a SQL database.
- Use those auto-generated methods to fetch and store data in a SQL database.

What Is An ORM?

An *Object Relational Mapping* is a library that allows you to access data stored in a SQL database through object-oriented, non-SQL code (such as JavaScript). You will write *object-oriented* code that accesses data stored in a *relational* SQL database like Postgres. The ORM is the *mapping* that will "translate" your object-oriented code into SQL code that the database can run. The ORM will automatically generate SQL code for common tasks like fetching and storing data.

You will learn how to use the Sequelize ORM.
Sequelize is the most widely used JavaScript ORM library.

How To Install Sequelize

After creating a new node project with `npm init` we are ready to install the Sequelize library.

```
npm install sequelize@^5.0.0
npm install sequelize-cli@^5.0.0
npm install pg@^8.0.0
```

We have installed not only the Sequelize library, but also a command line tool called `sequelize-cli` that will help us auto-generate and manage JavaScript files which will hold our Sequelize ORM code.

Last, we have also installed the pg library. This library allows Sequelize to access a Postgres database. If you were using a different database software (such as MySQL), you would need to install a different library.

How To Initialize Sequelize

We can run the command `npx sequelize init` to automatically setup the following directory structure for our project:

```
.
├── config
│   └── config.json
├── migrations
├── models
│   └── index.js
├── node_modules
├── package-lock.json
├── package.json
└── seeders
```

> Aside: the `npx` tool allows you to easily run scripts provided by packages like `sequelize-cli`. If you don't already have `npx`, you can install it with `npm install npx --global`. Without `npx` you would have

> to run the bash command: `./node_modules/.bin/sequelize init`. This directly runs the `sequelize` script provided by the installed `sequelize-cli` package.

Having run `npx sequelize init`, we must write our database login information into `config/config.json`.

By default this file contains different sections we call "environments". In a typical company you will have different database servers and configuration depending on where you app is running. Development is usually where you do your development work. In our case this is our local computer. But test might be and environment where you run tests, and production is the environment where real users are interacting with your application.

Since we are doing development, we can just modify the "development" section to look like this:

```
{
  "development": {
    "username": "catsdbuser",
    "password": "catsdbpassword",
    "database": "catsdb",
    "host": "127.0.0.1",
    "dialect": "postgres"
  }
}...
```

Here we are supposing that we have already created a `catsdb` database owned by the user `catsdbuser`, with password `catsdbpassword`. By setting `host` to `127.0.0.1`, we are saying that the database will run on the same machine as my JavaScript application. Last, we specify that we are using a `postgres` database.

Verifying That Sequelize Can Connect To The Database

At the top level of our project, we should create an `index.js` file. From this file we will verify that Sequelize can connect to the SQL database. To do this, we use the `authenticate` method of the sequelize object.

```
// ./index.js

const { sequelize } = require("./models");

async function main() {
  try {
    await sequelize.authenticate();
  } catch (e) {
    console.log("Database connection failure.");
    console.log(e);
    return;
  }

  console.log("Database connection success!");
  console.log("Sequelize is ready to use!");

  // Close database connection when done with it.
  await sequelize.close();
}

main();

// Prints:
//
// Executing (default): SELECT 1+1 AS result
// Database connection success!
// Sequelize is ready to use!
```

You may observe that the `authenticate` method returns a JavaScript `Promise` object. We use `await` to wait for the database connection to

be established. If `authenticate` fails to connect, the `Promise` will be rejected. Since we use `await`, an exception will be thrown.

Many Sequelize methods return `Promise`s. Using `async` and `await` lets us use Sequelize methods as if they were synchronous. This helps reduce code complexity significantly.

Note that I call `sequelize.close()`. This closes the connection to the database. A Node.js JavaScript program will not terminate until all open files and database connections are closed. Thus, to make sure the Node.js program doesn't "hang" at the end, we close the database connection. Otherwise we will be forced to kill the Node.js program with `CTRL-C`, which is somewhat annoying.

Our Preexisting Database Schema

We are assuming that we are working with a preexisting SQL database. Our `catsdb` has a single table: `Cats`. Using the `psql` command-line program, we can describe the pre-existing `Cats` table below.

```
catsdb=> \d "Cats"
                                      Table "public.Cats"
    Column    |           Type           | Collation | Nullable |              Default
--------------+--------------------------+-----------+----------+-------------------------------
 id           | integer                  |           | not null | nextval('"Cats_id_seq"'::regc
 firstName    | character varying(255)   |           |          |
 specialSkill | character varying(255)   |           |          |
 age          | integer                  |           |          |
 createdAt    | timestamp with time zone |           | not null |
 updatedAt    | timestamp with time zone |           | not null |
Indexes:
    "Cats_pkey" PRIMARY KEY, btree (id)
```

Besides a primary key `id`, each `Cats` record has a `firstName`, a `specialSkill`, and an `age`. Each record also keeps track of two timestamps: the time when the cat was created (`createdAt`), and the most recent time when a column of the cat has been updated (`updatedAt`).

Using Sequelize To Generate The Model File

We will configure Sequelize to access the `Cats` table via a JavaScript class called `Cat`. To do this, we first use our trusty Sequelize CLI:

```
# Oops, forgot age:integer! (Don't worry we'll fix it later)
npx sequelize model:generate --name Cat --attributes "firstName:string,specialSki
```

This command generates two files: a *model* file (`./models/cat.js`) and a *migration* file (`./migrations/20200203211508-Cat.js`). We will ignore the migration file for now, and focus on the model file.

When using Sequelize's `model:generate` command, we specify two things. First: we specify the *singular* form of the `Cats` table name (`Cat`). Second: we list the columns of the `Cats` table after the `--attributes` flag: `firstName` and `specialSkill`. We tell Sequelize that these are both `string` columns (Sequelize calls SQL `character varying(255)` columns `string`s).

We do not need to list `id`, `createdAt`, or `updatedAt`. Sequelize will always presume those exist. Notice that we have **forgotten** to list `age:integer` -- we will fix that soon!

Examining (And Modifying) A Sequelize Model File

Let us examine the generated `./models/cat.js` file:

```
// ./models/cat.js

'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: DataTypes.STRING,
    specialSkill: DataTypes.STRING
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

This file exports a function that defines a `Cat` class. When you use `Sequelize` to query the `Cats` table, each row retrieved will be transformed by Sequelize into an instance of the `Cat` class. A JavaScript class like `Cat` that corresponds to a SQL table is called a *model* class.

The `./models/cat.js` will not be loaded by us directly. Sequelize will load this file and call the exported function to define the `Cat` class. The exported function uses Sequelize's `define` method to auto-generate a new class (called `Cat`).

> Note: You may notice we aren't using the JavaScript's `class` keyword to define the Cat class. With Sequelize, it is going to do all that for us with the `define` method. This is because Sequelize was around way before the `class` keyword was added to JavaScript. It is possible to use the class keyword with Sequelize, but it's undocumented.

The first argument of `define` is the name of the class to define: `Cat`. Notice how the second argument is an `Object` of `Cats` table columns:

```
{
    firstName: DataTypes.STRING,
    specialSkill: DataTypes.STRING
}
```

This object tells Sequelize about each of the columns of `Cats`. It maps each column name (`firstName`, `specialSkill`) to the type of data stored in the corresponding column of the `Cats` table. It is unnecessary to list `id`, `createdAt`, `updatedAt`, since Sequelize will already assume those exist.

We can correct our earlier mistake of forgetting `age`. We update the definition as so:

```
const Cat = sequelize.define('Cat', {
  firstName: DataTypes.STRING,
  specialSkill: DataTypes.STRING,
  age: DataTypes.INTEGER,
}, {});
```

A complete list of Sequelize datatypes can be found in the documentation.

Using The `Cat` Model To Fetch And Update SQL Data

We are now ready to use our `Cat` model class. When Sequelize defines the `Cat` class, it will generate instance and class methods needed to interact with the `Cats` SQL table.

As we mentioned before we don't require our `cats.js` file directly. Instead we require `./models` which loads the file `./models/index.js`.

Inside this file it reads through all our models and attaches them to an object that it exports. So we can use destructuring to get a reference to our model class `Cat` like so:

```
const { sequelize, Cat } = require("./models");
```

Now let's update *our* `index.js` file to fetch a `Cat` from the `Cats` table:

```
const { sequelize , Cat } = require("./models");

async function main() {
  try {
    await sequelize.authenticate();
  } catch (e) {
    console.log("Database connection failure.");
    console.log(e);
    return;
  }

  console.log("Database connection success!");
  console.log("Sequelize is ready to use!");

  const cat = await Cat.findByPk(1);
  console.log(cat.toJSON());

  await sequelize.close();
}

main();

// This code prints:
//
// Executing (default): SELECT "id", "firstName", "specialSkill", "age", "created
// {
//   id: 1,
//   firstName: 'Markov',
//   specialSkill: 'sleeping',
//   age: 5,
//   createdAt: 2020-02-03T21:32:28.960Z,
//   updatedAt: 2020-02-03T21:32:28.960Z
// }
```

We use the `Cat.findByPk` static class method to fetch a single cat: the one with `id` equal to 1. This static method exists because our `Cat` model class *extends* `Sequelize.Model`.

"Pk" stands for *primary key*; the `id` field is the primary key for the `Cats` table. `findByPk` returns a `Promise`, so we must `await` the result. The result is an instance of the `Cat` model class.

The cleanest way to log a fetched database record is to first call the `toJSON` method. `toJSON` converts a `Cat` object to a *Plain Old JavaScript Object* (POJO). `Cat` instances have many private variables and methods that can be distracting when printed. When you call `toJSON`, only the public data fields are copied into a JavaScript `Object`. Printing this raw JavaScript `Object` is thus much cleaner.

> The author has a pet-peeve about the `.toJSON()` method of Sequelize, it does not return JSON. It instead returns a POJO. If you needed it to be JSON you would still need to call `JSON.stringify(cat.toJSON())`. Perhaps they should have called it `.toObject` or `.toPOJO` instead.

Note that Sequelize has logged the SQL query it ran to fetch Markov's information. This logging information is often helpful when trying to figure out what Sequelize is doing.

You'll also notice that Sequelize puts double quotes around the table and field names. So if you are trying to look at your "Cats" table from the `psql` command you will need to quote them there as well. This is because PostgreSQL lowercases all identifiers like table and fields names before the query is run if they aren't quoted.

Reading And Changing Record Attributes

While `toJSON` is useful for logging a `Cat` object, it is not the simplest way to access individual column values. To read the `id`, `firstName`, etc of a `Cat`, you can directly access those attributes on the `Cat` instance itself:

```
async function main() {
  // Sequelize authentication code from above...

  const cat = await Cat.findByPk(1);
  console.log(`${cat.firstName} has been assigned id #${cat.id}.`);
  console.log(`They are ${cat.age} years old.`)
  console.log(`Their special skill is ${cat.specialSkill}.`);

  await sequelize.close();
}

main();

// This code prints:
//
// Executing (default): SELECT "id", "firstName", "specialSkill", "age", "created
// Markov has been assigned id #1.
// They are 5 years old.
// Their special skill is sleeping.
```

Accessing data directly through the `Cat` object is just like reading an attribute on any other JavaScript class. You may likewise *change* values in the database:

```
async function main() {
  // Sequelize authentication code from above...

  // Fetch existing cat from database.
  const cat = await Cat.findByPk(1);
  // Change cat's attributes.
  cat.firstName = "Curie";
  cat.specialSkill = "jumping";
  cat.age = 123;

  // Save the new name to the database.
  await cat.save();

  await sequelize.close();
}

// Prints:
//
// Executing (default): SELECT "id", "firstName", "specialSkill", "age", "created
// Executing (default): UPDATE "Cats" SET "firstName"=$1,"specialSkill"=$2,"age"=

main();
```

Note that changing the `firstName` attribute value does not immediately change the stored value in the SQL database. Changing the `firstName` without calling `save` **has no effect** on the database. Only when we call `cat.save()` (and `await` the promise to resolve) will the changes to `firstName`, `specialSkill`, and `age` be saved to the SQL database. All these values are updated simultaneously.

Conclusion

Having completed this reading, you should be able to:

- Describe what an Object Relational Mapping is and what it is used for.
- Install the `sequelize`, `sequelize-cli`, `pg` packages.
- Configure Sequelize via the `config/config.json` file.
- Use Sequelize's `authenticate` method to verify that Sequelize can connect to the database.
- Use the Sequelize CLI `model:generate` command to generate a model file.
- Configure a model file to tell Sequelize about each database column.
- Use the `findByPk` class method to fetch data from a SQL table.
- Read data attributes from a model instance.
- Modify a model instance's attributes and save the changes back to the SQL database using the `save` method.

Using Database Migrations

We've seen how to use an ORM like Sequelize to fetch and store data in a SQL database using JavaScript classes and methods. Sequelize also lets you write JavaScript code that creates, modifies, or drops SQL tables. The JavaScript code that does this is called a *migration*. A migration "moves" the database from an old schema to a new schema.

When you finish this reading you will be able to:

- Describe advantages to using migrations over raw SQL commands to create and drop tables.
- Write migrations that create and drop tables.
- Undo incorrect migrations, fix them, and rerun them.

Sequelize Migration Files

In the prior reading we assumed that a `Cats` table already existed in our `catsdb` database. In this reading, we will presume that the `Cats`

table does not exist, and that we have to create the table ourselves. This is the typical case when you aren't merely interacting with a preexisting database. When you develop your own application, the database will start out empty and with a blank schema.

We previously used the Sequelize CLI tool to autogenerate a `Cat` model file like so:

```
# Oops, forgot age:integer!
npx sequelize model:generate --name Cat --attributes "firstName:string,specialSki
```

We noted that this creates *two* files. We've already examined the model file `./models/cat.js`. We will now look at the auto-generated *migration* file `./migrations/20200203211508-create-cat.js`.

```
// ./migrations/20200203211508-create-cat.js

'use strict';
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Cats', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      firstName: {
        type: Sequelize.STRING
      },
      specialSkill: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Cats');
  }
};
```

The migration file exports two functions: `up` and `down`. The `up` function tells Sequelize how to create a `Cats` table. The `down` function tells Sequelize how to "undo" the `up` function. The `down` function drops the `Cats` table.

We will examine these functions more closely, but let's first see how to use a migration.

> Note: The timestamp `20200203211508` preceding `-create-cat.js` represents February 2, 2020. It gives the time and day that the migration was generated. (Your date should be when you generated your migration)
> By using the date and time as part of the filename, all migration files will have unique names. Also, alphabetical sorting will order the files from oldest to most recent migration.

Running A Migration

To create the `Cats` table, we must run our migration code. Having generated the `20200203211508-create-cat.js` migration file, we will use the Sequelize CLI tool to run the migration. We may do this like so:

```
# Run the migration's `up` method.
npx sequelize db:migrate
```

By giving Sequelize the `db:migrate` subcommand, it will know that we are asking it to run any new migrations. To run a migration, Sequelize will call the `up` method defined in the migration file. The `up` method will run the necessary `CREATE TABLE ...` SQL command for us. Sequelize will record (in a special `catsdb` table called `SequelizeMeta`) that the migration has been run. The next time we call `npx sequelize db:migrate`, Sequelize will not try to "redo" this already performed migration. It will do nothing the second time.

Having run the migration, we can verify that the `Cats` table looks like it should (with the exception of the `age` column):

Note that we we are using the table name in quotes here in `psql`.

```
catsdb=> \d "Cats";
                                        Table "public.Cats"
    Column    |           Type           | Collation | Nullable
--------------+--------------------------+-----------+----------
 id           | integer                  |           | not null
 firstName    | character varying(255)   |           |
 specialSkill | character varying(255)   |           |
 createdAt    | timestamp with time zone |           | not null
 updatedAt    | timestamp with time zone |           | not null
Indexes:
    "Cats_pkey" PRIMARY KEY, btree (id)
```

Rolling Back A Migration

We made a mistake when generating our `Cat` migration. We forgot to include the `age` column.

One way to fix this is to generate a *second* migration that adds the forgotten `age` column. If we have already pushed our migration code to a remote git server, we should opt for this option.

If the migration has not yet been pushed, we can fix the migration directly. We will "undo" (AKA *rollback*) the migration that created the `Cats` table (dropping the table), fix the `up` method so that the `age` column is included, and finally rerun the migration.

> Note: this is not the same as the SQL command ROLLBACK.

To undo the migration, we run:

```
npx sequelize db:migrate:undo
```

Sequelize will call the `down` method for us, and the `Cats` table is dropped.

Why should you not use the `db:migrate:undo` way when the migration file has already been pushed to a remote git server? The reason is this: you can easily tell other developers to fix a broken migration by writing a second fixup migration (for instance, that adds the `age` column). All you need to do is check this new migration file into source control and push it. When another developer pulls your new migration code, the next time they run `npx sequelize db:migrate`, your fixup migration will be run on their local machine.

When rolling back already-checked-in migrations, there is no way to easily communicate to other developers that they should (1) rollback your migration and (2) rerun the newly corrected version of this migration. To avoid this communication problem, you should only rollback commits if you haven't already pushed them to a remote git server.

Editing A Migration File

Let's examine the `up` and `down` methods more closely. Let's start with the `up` method:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Cats', {
      id: {
        allowNull: false,
        autoIncrement: true,
        primaryKey: true,
        type: Sequelize.INTEGER
      },
      firstName: {
        type: Sequelize.STRING
      },
      specialSkill: {
        type: Sequelize.STRING
      },
      createdAt: {
        allowNull: false,
        type: Sequelize.DATE
      },
      updatedAt: {
        allowNull: false,
        type: Sequelize.DATE
      }
    });
  },
  // ...
};
```

The `up` method will be passed a QueryInterface (documentation) object. This object provides a number of commands for modifying the SQL database schema. The `createTable` method is amongst the most important.

We pass the table name (`'Cats'`) along with an object mapping column names to column attributes. Every column must have a specified `type`. This is similar to what we saw when we generated a model file. Note that we **do not** take `id`, `createdAt`, or `updatedAt` for granted. We

need to include those columns. Luckily, everything has been auto-generated for us!

We will talk about `allowNull` and `primaryKey` in a later reading. These attributes ask Sequelize to add database constraints to a column. Likewise we will ignore `autoIncrement` for the moment (this allows a unique `id` to be auto-generated by the database for each saved row in the `Cats` table).

We fix the `up` method like so:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    return queryInterface.createTable('Cats', {
      // ...
      firstName: {
        type: Sequelize.STRING
      },
      specialSkill: {
        type: Sequelize.STRING
      },
      // Here we add the `age` column.
      age: {
        type: Sequelize.INTEGER,
      },
      // ...
    });
  },
  // ...
};
```

Adding the `age` column to the migration is a lot like how we added `age` to our model file.

Having fixed our migration, we may now "rerun" it the same way we ran it the first time:

```
npx sequelize db:migrate
```

We may now behold the fixed table:

```
catsdb=> \d "Cats"
                                          Table "public.Cats"
    Column    |           Type           | Collation | Nullable
--------------+--------------------------+-----------+----------
 id           | integer                  |           | not null
 firstName    | character varying(255)   |           |
 specialSkill | character varying(255)   |           |
 age          | integer                  |           |
 createdAt    | timestamp with time zone |           | not null
 updatedAt    | timestamp with time zone |           | not null
Indexes:
    "Cats_pkey" PRIMARY KEY, btree (id)
```

`up` And `down` are Asynchronous

A final note about `up` (and also `down`). Sequelize expects `up` to be *asynchronous*. That is, Sequelize expects `up` to return a `Promise` object. Sequelize will wait for the `Promise` to be resolved. When the `Promise` is resolved, Sequelize will know the work of the `up` method is complete.

The `createTable` method is also asynchronous (returns a `Promise`). The promise resolves when `createTable` is done creating the table. This is why `up` is written as:

```
module.exports = {
  up: (queryInterface, Sequelize) => {
    // up returns Promise returned by `createTable`.
    return queryInterface.createTable('Cats', {
      // ...
    });
  },
  // ...
};
```

Sequelize is able to autogenerate a migration to create a `Cats` table, but many other migrations (for instance, to add an `age` column to our `Cats` table) must be written by hand. When writing your own migrations, you may prefer using `async` / `await`, which is clearer:

```
module.exports = {
  // Note the addition of the `async` keyword
  up: async (queryInterface, Sequelize) => {
    // await `createTable` to finish its work.
    await queryInterface.createTable('Cats', {
      // ...
    });

    // No need to return anything. An `async` method always returns a

    // Promise that waits for all `await`ed work to be performed.
  },
  // ...
};
```

Writing A `down` Method

A `down` method is written just like an `up` method. In the `down` method we "undo" what has been performed by the `up` method. We call `QueryInterface`'s `dropTable` method to drop the `Cats` table we created in `up`:

```
module.exports = {
  // ...
  down: (queryInterface, Sequelize) => {
    return queryInterface.dropTable('Cats');
  }
};

// OR, async/await way:
module.exports = {
  // ...
  down: async (queryInterface, Sequelize) => {
    await queryInterface.dropTable('Cats');
  }
};
```

Imagine we had forgotten to drop the `Cats` table in the `down` method. That is: imagine the `down` method was somehow left empty. If we rollback the migration nothing will be done by the empty `down` method. Thus the incorrect `Cats` table we created will not have been dropped. The wrong `Cats` table would still exist.

Imagine we next fix the migration's `up` method. We want to rerun the migration now and create the corrected `Cats` table. But when try to do this, Sequelize will hit an error! Rerunning the migration will try to `CREATE TABLE "Cats"` again, but SQL will complain because a `Cats` table already exists. It was created the first time we ran the migration, but never dropped when we tried to rollback the migration!

Inevitably all programmers will sometimes make mistakes like this. In these circumstances, you will probably have to open `psql` and write a SQL `DROP TABLE` command to fix things. Having manually corrected things, you can finally rerun the corrected migration.

You should **never** manually drop a table on a production database. That is incredibly dangerous, and typically cannot be undone. Even if database backups do not exist, recently inserted data will be lost

forever. This is yet another reason why you ought not rollback migrations that have been pushed from your local development environment!

Advantages Of Migrations

Having seen how to *use* Sequelize migrations, we can discuss their benefits versus writing SQL commands like `CREATE TABLE ...` yourself.

The first advantage is that Sequelize migration code is written in JavaScript, which you may find simpler to write/read than the corresponding SQL code. Most programmers write more JavaScript than SQL, so they are typically better at remembering how to do things in JavaScript than in SQL.

A second advantage is that migration files store SQL schema change code permanently. The migration files can be checked into git, so that you don't ever forget how your database was configured.

A third (related) advantage comes when another developer wants to collaborate on your JavaScript program. By cloning your git repository, they get all the migration files, too. To setup their own copy of your database, a collaborator can run the migration files on their own computer, playing back the schema changes one-by-one. Because they apply the same migrations as you, they end up with the same schema as you.

Last, by using migrations you are able to rollback database changes to fix bugs. This can be helpful in a local development environment where it is typical to make mistakes. Remember though: you should **never** rollback migrations that have been run on a production server.

Conclusion

Having completed this reading, you now are able to:

- Describe advantages to using migrations over raw SQL commands to create, modify, and drop tables.
- Generate (and modify as needed)) migrations that create and drop tables.
- Run migrations to change the database schema.
- Undo incorrect migrations, fix them, and rerun them.

CRUD Operations Using Sequelize

There are four general ways to interact with a database. To illustrate these, recall our `Cats` table. We can:

1. Save a new cat to the database by *creating* a new row in the `Cats` table,
2. We can *read* previously stored cat data by fetching a row (or multiple rows) out of the `Cats` table,
3. We can *update* some of the column values for a pre-existing cat by modifying a row in the `Cats` table,
4. We can delete (*destroy*) the data for a cat by removing a row in the `Cats` table.

These four actions are sometimes abbreviated as *CRUD*. After this reading, you will be able to:

- Use Sequelize to create new records in a table,
- Use Sequelize to read/fetch existing records by primary key,
- Use Sequelize to update existing records with new attribute values,
- Use Sequelize to delete records from a table.

Creating A New Record

To save a new cat's data as a row in the `Cats` table, we do a two step process:

1. We call the static `build` method on the `Cat` class with the desired values.
2. We call the `save` method on the `cat` instance.

Let's see an example:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Constructs an instance of the JavaScript `Cat` class. **Does not
  // save anything to the database yet**. Attributes are passed in as a
  // POJO.
  const cat = Cat.build({
    firstName: "Markov",
    specialSkill: "sleeping",
    age: 5,
  });

  // This actually creates a new `Cats` record in the database. We must
  // wait for this asynchronous operation to succeed.
  await cat.save();

  console.log(cat.toJSON());

  await sequelize.close();

}

main();
```

Running the code:

```
Executing (default): INSERT INTO "Cats" ("id","firstName","specialSkill","age","createdAt","upd
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'sleeping',
  age: 5,
  updatedAt: 2020-02-11T19:04:23.116Z,
  createdAt: 2020-02-11T19:04:23.116Z
}
```

A new row has been inserted into the `Cats` table. We see that `id`, `updatedAt`, and `createdAt` were each autogenerated for us.

Reading A Record By Primary Key

Let's read an existing record from the database:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch the cat with id #1.
  const cat = await Cat.findByPk(1);
  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

Running this code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'sleeping',
  age: 5,
  createdAt: 2020-02-11T19:04:23.116Z,
  updatedAt: 2020-02-11T19:04:23.116Z
}
```

Fetching a record by primary key is the most common form of read operation from a database. In another reading we will learn other ways to fetch data. For instance: we will learn how to fetch all cats named "Markov" (there may be many).

Updating A Record

Let's tweak our reading code to change (*update*) an attribute of Markov:

```
const { sequelize, Cat }  = require("./models");

async function main() {
  const cat = await Cat.findByPk(1);

  console.log("Old Markov: ");
  console.log(cat.toJSON());

  // The Cat object is modified, but the corresponding record in the
  // database is *not* yet changed at all.
  cat.specialSkill = "super deep sleeping";
  // Only by calling `save` will the data be saved.
  await cat.save();

  console.log("New Markov: ");
  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

Running this code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
Old Markov:
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'sleeping',
  age: 5,
  createdAt: 2020-02-11T19:04:23.116Z,
  updatedAt: 2020-02-11T19:04:23.116Z
}
Executing (default): UPDATE "Cats" SET "specialSkill"=$1,"updatedAt"=$2 WHERE "id" = $3
New Markov:
{
  id: 1,
  firstName: 'Markov',
  specialSkill: 'super deep sleeping',
  age: 5,
  createdAt: 2020-02-11T19:04:23.116Z,
  updatedAt: 2020-02-11T19:15:08.668Z
}
```

Important note: changing an attribute of a `Cat` object does not immediately change any data in the `Cats` table. To change data in the `Cats` table, you must also call `save`. If you forget to call `save`, no data will be changed. `save` is asynchronous, so you must also `await` for it to complete.

If you look carefully, you can see that the `updatedAt` attribute was changed for us when we updated Markov!

Destroying A Record

We can also destroy records and remove them from the database:

```
const process = require("process");

const { sequelize , Cat } = require("./models");

async function main() {
  const cat = await Cat.findByPk(1);
  // Remove the Markov record.
  await cat.destroy();

  await sequelize.close();
}

main();
```

This code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
Executing (default): DELETE FROM "Cats" WHERE "id" = 1
```

Class Methods For CRUD

When creating a record, you can avoid the two step process of (1) creating a `Cat` instance and (2) calling the `save` instance method. You can do a one step process of calling the `create` **class method**:

```
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = await Cat.create({
    firstName: "Curie",
    specialSkill: "jumping",
    age: 4,
  });

  console.log(cat.toJSON());

  await sequelize.close();
}

main();
```

The `create` class method does both steps in one. It is just a convenience. Similar to before, this code prints:

```
Executing (default): INSERT INTO "Cats" ("id","firstName","specialSkill","age","createdAt","upd
{
  id: 3,
  firstName: 'Curie',
  specialSkill: 'jumping',
  age: 4,
  updatedAt: 2020-02-11T19:36:03.858Z,
  createdAt: 2020-02-11T19:36:03.858Z
}
```

When destroying, we also did a two step process: (1) fetch the record, (2) call the `destroy` instance method. Instead, we could just call the `destroy` **class method** directly:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Destroy the Cat record with id #3.
  await Cat.destroy({ where: { id: 3 } });

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): DELETE FROM "Cats" WHERE "id" = 3
```

An advantage to the class method form of destroying is that we avoid an unnecessary fetch of `Cat.findByPk(3)`. Database queries can sometimes be slow, though typically a few extra queries won't make a big difference. Choosing between the instance and class methods of destroying usually comes down to which you consider easier to read/understand.

Conclusion

As ever, the best resource for learning about Sequelize model methods is the documentation. The documentation explains the `create`, `destroy`, `findByPk`, and `save` methods in depth.

Having completed this reading, you now know how to:

- Use Sequelize to create new records in a table (using both instance and class methods),
- Use Sequelize to read/fetch existing records by primary key,

- Use Sequelize to update existing records with new attribute values,
- Use Sequelize to delete records from a table (using both instance and class methods).

Querying Using Sequelize

We have already seen how to find a single record by primary key: `findByPk`. In this reading we will learn about more advanced ways to query a table. We will learn how to:

- Fetch all `Cats` whose name is `"Markov"`,
- Fetch all `Cats` whose name is `"Markov"` **OR** `"Curie"`,
- Fetch all `Cats` whose age is **not** 5,
- Fetch all `Cats` whose name is `"Markov"` **AND** whose age is 5,
- Fetch all `Cats` whose age is **less than** 5,

We will also learn how to:

- Order `Cats` results by age (descending or ascending),
- Limit `Cats` results to a finite number.

Basic Usage Of `findAll` To Retrieve Multiple Records

Let's consider a simple example where we want to retrieve all the `Cats` in the database:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // `findAll` asks to retrieve _ALL_ THE CATS!!  An array of `Cat`
  // objects will be returned.
  const cats = await Cat.findAll();

  // Log the fetched cats.
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

Since this is an array we can't use that `.toJSON()` method we learned earlier, so we can instead use `JSON.stringify` on the Array.

Pro tip: giving a 3rd argument to `JSON.stringify` will pretty-print the result with the specified spacing. (We pass `null` as the 2nd argument to skip it.) You can read more at the `JSON.stringify` docs.

Running this code prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

It isn't typical to want to fetch *every* record. We typically want to get only those records that match some criterion. In SQL, we use a `WHERE` clause to do this. With Sequelize, we issue a `WHERE` query like so:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch all cats named Markov.
  const cats = await Cat.findAll({
    where: {
      firstName: "Markov",
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

Which prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  }
]
```

We've passed the `findAll` class method the `where` option. The `where` option tells Sequelize to use a `WHERE` clause. The option value passed is `{ firstName: "Markov" }`. This tells Sequelize to only return those `Cats` where `firstName` is equal to `"Markov"`.

If we wanted to select those `Cats` named Markov **OR** Curie, we can map `firstName` to an array of `["Markov", "Curie"]`. For example:

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch all cats named either Markov or Curie.
  const cats = await Cat.findAll({
    where: {
      firstName: ["Markov", "Curie"],
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

The difference is that we've passed `{ firstName: ["Markov", "Curie" ]}`. Sequelize will return all `Cats` whose `firstName` matches either `"Markov"` or `"Curie"`.

Using `findAll` To Find Objects Not Matching A Criterion

We can also find all the `Cats` whose names are **NOT** Markov, but we will need to require in the `Op` object from Sequelize so we can use the "not equal" operator from it:

```
const { Op } = require("sequelize");
const { sequelize, Cat } = require("./db/models");

async function main() {
  const cats = await Cat.findAll({
    where: {
      firstName: {
        // Op.ne means the "not equal" operator.
        [Op.ne]: "Markov",
      },
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

Prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

This is our first example of a Sequelize *operator*: `Op.ne`. `ne` stands for "not equal." Instead of mapping `firstName` to a single value like `"Markov"` or an array of values like `["Markov", "Curie"]`, we have mapped it to:

```
{ [Op.ne]: "Markov" }
```

How does this work? `Op.ne` is a JavaScript *symbol*: `Op.ne === Symbol.for('ne')`. To simplify, let's just imagine that `Op.ne === "ne"`.

When we write `{ [Op.ne]: "Markov" }`, the `[]` brackets perform key interpolation. So this is equal to `{ "ne": "Markov" }`. So overall, we are effectively writing:

```
db.Cat.findAll({
  where: {
    // Won't exactly work (you need to use `[Op.ne]` after all). Does
    // illustrate the concept though.
    firstName: { "ne": "Markov" },
  },
})
```

This perhaps makes it clearer how Sequelize understands what we want. Sequelize is being passed an *object* as the `firstName` value. The object is specifying that we want to do a `!=` SQL operation by using the `"ne"` ("not equal") key. The value to "not equal" is specified as `"Markov"`.

Combining Criteria with `Op.and`

We've seen one way to do an `OR` operation above (by mapping a column name to an array of values). Let's see how to do an `AND` operation:

```
const { Op } = require("sequelize");
const { sequelize , Cat } = require("./models");

async function main() {
  // fetch cats with name != Markov AND age = 4.
  const cats = await Cat.findAll({
    where: {
      firstName: {
        [Op.ne]: "Markov",
      },
      age: 4,
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

Simply by listing more key/value pairs in the `where` object, we ask Sequelize to "AND" together multiple criteria.

Another way to do the same thing is like so:

```
const { Op } = require("sequelize");
const { sequelize , Cat } = require("./models");

async function main() {
  const cats = await db.Cat.findAll({
    where: {
      [Op.and]: [
        { firstName: { [Op.ne]: "Markov" } },
        { age: 4 },
      ],
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

The use of the `Op.and` operator is somewhat similar to `Op.ne`. This time we map `Op.and` to an *array* of criteria. Returned records must match all the criteria.

Combining Criteria with `Op.or`

We've already seen how to do an `OR` to match a *single column* against *multiple values*. You can use `Op.or` for even greater flexibility:

```
const { Op } = require("sequelize");
const { sequelize, Cat } = require("./models");

async function main() {
  // fetch cats with name == Markov OR age = 4.
  const cats = await Cat.findAll({
    where: {
      [Op.or]: [
        { firstName: "Markov" },
        { age: 4 },
      ],
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

Our query is to find all cats whose names are "Markov" and whose age is 4. Therefore both cats are returned: Markov and Curie (whose age is 4).

Querying With Comparisons

We can use operators like `Op.gt` (greater than) and `Op.lt` (less than) to select by comparing values. We use these just like `Op.ne`:

```
const { Op } = require("sequelize");
const { sequelize, Cat } = require("./models");

async function main() {
  // Fetch all cats whose age is > 4.
  const cats = await Cat.findAll({
    where: {
      age: { [Op.gt]: 4 },
    },
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  }
]
```

Ordering Results

We've seen how to use a `where` query option to filter results with a SQL `WHERE` clause. We can use the `order` query option to perform a SQL `ORDER BY`:

```
const { sequelize, Cat } db = require("./models");

async function main() {
  const cats = await Cat.findAll({
    order: [["age", "DESC"]],
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This prints:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  },
  {
    "id": 5,
    "firstName": "Curie",
    "specialSkill": "jumping",
    "age": 4,
    "createdAt": "2020-02-11T23:03:25.398Z",
    "updatedAt": "2020-02-11T23:03:25.398Z"
  }
]
```

We've specified `{ order: [["age", "DESC"]] }`. Notice how we specify the sort order with a doubly-nested array. If we wanted to sort ascending we could more simply write: `{ order: ["age"] }`.

What if we wanted to sort by *two* columns? For instance, say we wanted to `SORT BY age DESC, firstName`. We would write: `{ order: [["age", "DESC"], "firstName"] }`. That would sort descending by `age`, and then ascending by `firstName` for cats with the same age.

Limiting Results and `findOne`

If we want only the oldest cat we can use the `limit` query option:

```
const { sequelize, Cat } = require("./models");

async function main() {
  const cats = await Cat.findAll({
    order: [["age", "DESC"]],
    limit: 1,
  });
  console.log(JSON.stringify(cats, null, 2));

  await sequelize.close();
}

main();
```

This selects only one (the oldest) cat:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
[
  {
    "id": 4,
    "firstName": "Markov",
    "specialSkill": "sleeping",
    "age": 5,
    "createdAt": "2020-02-11T23:03:25.388Z",
    "updatedAt": "2020-02-11T23:03:25.388Z"
  }
]
```

Since we know that there will be only one result, it is pointless to return an array. In cases when we want a maximum of one result, we can use `findOne`:

```
const { sequelize, Cat } = require("./models");

async function main() {
  const cat = await Cat.findOne({
    order: [["age", "DESC"]],
  });
  console.log(JSON.stringify(cat, null, 2));

  await sequelize.close();
}

main();
```

Which prints:

```
>> node index.js
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
{
  "id": 4,
  "firstName": "Markov",
  "specialSkill": "sleeping",
  "age": 5,
  "createdAt": "2020-02-11T23:03:25.388Z",
  "updatedAt": "2020-02-11T23:03:25.388Z"
}
```

This returned the `Cat` object directly, not wrapped in an array.

If there is no record matching the criteria passed to `findOne`, it will return `null` (rather than an empty array):

```
const { sequelize, Cat } = require("./models");

async function main() {
  // Try to find a non-existant cat.
  const cat = await Cat.findOne({
    where: {
      firstName: "Franklin Delano Catsevelt",
    },
  });
  console.log(JSON.stringify(cat, null, 2));

  await sequelize.close();
}

main();
```

No such cat exists:

```
Executing (default): SELECT "id", "firstName", "specialSkill", "age", "createdAt", "updatedAt"
null
```

Conclusion

We've scratched the surface of the many query options supported by Sequelize. You may find more information as necessary by reading the Sequelize querying documentation. You can in particular review the list of Sequelize query operators.

Now that you've completed this reading you should know how to:

- Use the `where` query option,
- Use the `Op.and` operator to match **all** of multiple criteria,
- Use the `Op.or` operator to match **any** of multiple criteria,
- Use the `Op.ne` to match rows where the value **does not equal** the specified value,
- Use the `Op.gt`, `Op.lt` operators to **compare** values,
- Use the `order` query option to **order** results,
- Use the `limit` query option to **limit** the number of returned results,
- Use `findOne` when only one result is expected or desired.

Model Validations With Sequelize

It's important to make sure that data stored to a database is not erroneous or incomplete. Imagine the following forms of "garbage data:"

- A `Cats` record with `firstName` set to `NULL`. All `Cats` ought to have a name.
- A `Cats` record with `firstName` set to the empty string: `""`.
- A `Cats` record with an `age` less than `0`. `age` must always be non-negative.
- Perhaps the `specialSkill` should come from a pre-defined limited list of `["jumping", "sleeping", "purring"]`. A `Cats` record with a `specialSkill` of `"pearl diving"` would thus be invalid.

Sequelize lets us write JavaScript code that will check that these data requirements are satisfied before saving a record to the database. The JavaScript code that does this is called a *validation*. A validation is code that makes sure that data is valid.

In this reading you will learn how to:

1. Validate that an attribute is not set to `NULL`.
2. Validate that a string attribute is not set to the empty string `""`.
3. Validate that a string attribute is not too long (has too many characters).
4. Validate that a numeric attribute meets minimum or maximum thresholds.
5. Validate that an attribute is within a limited set of options.

Validating That An Attribute Is Not `NULL`

We should not allow a `Cat` to be saved to the database if it lacks

1. a `firstName`,
2. an `age`, or
3. a `specialSkill`.

None of these should be set to `NULL`.

Before adding validations to check these requirements, let's review what our `Cat` model code currently looks like:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: DataTypes.STRING,
    specialSkill: DataTypes.STRING,
    age: DataTypes.INTEGER,
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

We will modify our model definition to give more specific instructions to Sequelize about the `firstName`, `specialSkill`, and `age` attributes:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "firstName must not be null",
        },
      },
    },
    specialSkill: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "specialSkill must not be null",
        },
      },
    },
    age: {
      type: DataTypes.INTEGER,
      allowNull: false,
      validate: {
        notNull: {
          msg: "age must not be null",
        },
      },
    },
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

What has changed? We now map each attribute name (`firstName`, `specialSkill`, `age`) to a POJO that tells Sequelize how to configure

that attribute. Here is the POJO for `firstName`:

```
{
  type: DataTypes.STRING,
  allowNull: false,
  validate: {
    notNull: {
      msg: "firstName must not be null",
    },
  },
}
```

The `type` attribute is of course vital: this used to be the only thing we specified. We've added two new attributes. The first is `allowNull: false`. This tells Sequelize not to let us set the `firstName` attribute to `NULL`.

The second attribute is `validate`. We will spend a lot of time examining this attribute in this reading. Validation logic for `firstName` is configured inside the `validate` attribute. Our `validate` configuration is:

```
{
  notNull: {
    msg: "firstName must not be null",
  },
}
```

This configuration tells Sequelize what error message to give if we try to set the `firstName` attribute to `NULL`. It's odd that we have to set both `allowNull: false` and `notNull: { msg: ... }`. This feels like unnecessary duplication. Regardless, that's what Sequelize wants us to do. On the other hand, we do get a chance to specify the error message to print if the validation fails (`"firstName must not be null"`).

Let's see how the validation logic helps us avoid saving junk data to our database:

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

async function main() {
  const cat = Cat.build({
    // Empty cat. All fields set to `null`.
  });

  try {
    // Try to save cat to the database.
    await cat.save();

    console.log("Save success!");
    console.log(JSON.stringify(cat, null, 2));
  } catch (err) {
    console.log("Save failed!");

    // Print list of errors.
    for (const validationError of err.errors) {
      console.log("*", validationError.message);
    }
  }

  await sequelize.close();
}

main()
```

Running this code prints:

```
Save failed!
* firstName must not be null
* specialSkill must not be null
* age must not be null
```

What happened? When we call the `save` method on a `Cat`, Sequelize will check that all the specified validations are satisfied. In this case none of them are! The `save` method will throw an exception, which we handle using `try { ... } catch (err) { ... }`.

What kind of exception? The thrown error is a `ValidationError`. This has an `errors` attribute, which stores an array of `ValidationErrorItem`s. We print out the message for each item error.

Because there were validation failures, Sequelize **will not save** the invalid `Cats` record to the database. Sequelize thus keeps us from inserting junk data into the database.

If we want to save our `Cat` object, we would have to change its attributes to meet the validations (i.e., set them to something other than `NULL`) and call `save` a second time. For example:

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

async function main() {
  const cat = Cat.build({
    // Empty cat. All fields set to `null`.
  });

  try {
    await cat.save();
  } catch (err) {
    // The save will not succeed!
    console.log("We will fix and try again!");
  }

  // Fix the various validation problems.
  cat.firstName = "Markov";
  cat.specialSkill = "sleeping";
  cat.age = 4;

  try {
    // Trying to save a second time!
    await cat.save();

    console.log("Success!");
  } catch (err) {
    // The save *should* succeed!
    console.log(err);
  }

  await sequelize.close();
}

main()
```

The `notEmpty` Validation

Even though we are not allowed to set `firstName` and `specialSkill` to `NULL`, we could still set them to the empty string `""`:

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

async function main() {
  const cat = Cat.build({
    firstName: "",
    specialSkill: "",
    age: 5,
  });

  try {
    // Try to save cat to the database.
    await cat.save();

    console.log("Save success!");
    console.log(JSON.stringify(cat, null, 2));
  } catch (err) {
    console.log("Save failed!");

    // Print list of errors.
    for (const validationError of err.errors) {
      console.log("*", validationError.message);
    }
  }

  await sequelize.close();
}

main();
```

```
Executing (default): INSERT INTO "Cats" ("id","firstName","specialSkill","age","createdAt","upd
Save success!
{
  "id": 8,
  "firstName": "",
  "specialSkill": "",
  "age": 5,
  "updatedAt": "2020-02-12T21:34:49.250Z",
  "createdAt": "2020-02-12T21:34:49.250Z"
}
```

This is bogus: `Cats` records should have a non-empty `firstName` and `specialSkill`. We will therefore add a second validation for both `firstName` and `specialSkill`:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "firstName must not be null",
        },
        notEmpty: {
          msg: "firstName must not be empty",
        },
      },
    },
    specialSkill: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "specialSkill must not be null",
        },
        notEmpty: {
          msg: "specialSkill must not be empty",
        },
      },
    },
    // ...
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

When we run the same `index.js` that tries to save the `Cats` record with the empty `firstName` and `specialSkill`, we now print:

```
Save failed!
* firstName must not be empty
* specialSkill must not be empty
```

Excellent! We've added the new validation by adding a `notEmpty` key to the `validate` POJO. Just like with `notNull`, we specify a message to print.

This is the typical story: we add new validations by adding new key/value pairs to the `validate` POJO. Sequelize provides many different kinds of validations for us, but we configure all of them in the same general manner.

Forbidding Long String Values

We don't want our cats to have names that are too long. We add a `len` validation like so:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    firstName: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "firstName must not be null",
        },
        notEmpty: {
          msg: "firstName must not be empty",
        },
        len: {
          args: [0, 8],
          msg: "firstName must not be more than eight letters long",
        },
      },
    },
    // ...
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

If we try to run:

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

async function main() {
  const cat = Cat.build({
    firstName: "Markov The Magnificent",
    specialSkill: "sleeping",
    age: 5,
  });

  try {
    // Try to save cat to the database.
    await cat.save();

    console.log("Save success!");
    console.log(JSON.stringify(cat, null, 2));
  } catch (err) {
    console.log("Save failed!");

    // Print list of errors.
    for (const validationError of err.errors) {
      console.log("*", validationError.message);
    }
  }

  await sequelize.close();
}

main();
```

We will be told:

```
Save failed!
* firstName must not be more than eight letters long
```

The `len` validation gets a `msg` attribute as usual. We also configure `args: [0, 8]`. These are the "arguments" to the `len` validation. We are telling Sequelize to trigger a validation error if the `firstName` property has a length less than zero (impossible) or greater than eight.

Note that even though the `len` validation is not triggered for a length of zero, the `notEmpty` validation still will be.

If desired, we could use the `len` validation to set a true minimum length for a string. If we wanted a minimum length of two letters, we would just change `args: [2, 8]`. (We ought also update the `msg` appropriately.)

Validating That A Numeric Value Is Within A Specified Range

A `Cat` should never have a negative age. Perhaps, also, a `Cat` should have a theoretical maximum age of 99 years. We can add validations to enforce these requirements:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    // ...
    age: {
      type: DataTypes.INTEGER,
      allowNull: false,
      validate: {
        notNull: {
          msg: "age must not be null",
        },
        min: {
          args: [0],
          msg: "age must not be less than zero",
        },
        max: {
          args: [99],
          msg: "age must not be greater than 99",
        },
      },
    },
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

You can see that the `min` and `max` validations are configured in the same sort of way that the `len` validation is.

If we try to save a `Cat` with an `age` of `-1`, we are printed:

```
Save failed!
* age must not be less than zero
```

Likewise, if we try to save a `Cat` with an `age` of `123` we are printed:

```
Save failed!
* age must not be greater than 99
```

(Note: I've stopped repeating our `index.js` file, since there are only trivial modifications to a `Cat`'s attributes each time.)

Validating That An Attribute Is Among A Finite Set Of Values

Let's say that a `Cat`'s `specialSkill` should be restricted to a pre-defined list of `["jumping", "sleeping", "purring"]`. That is: a `Cat` should not be allowed to have just any `specialSkill`. The `specialSkill` must be on the list.

We can enforce this requirement like so:

```
// ./models/cat.js
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Cat = sequelize.define('Cat', {
    // ...
    specialSkill: {
      type: DataTypes.STRING,
      allowNull: false,
      validate: {
        notNull: {
          msg: "specialSkill must not be null",
        },
        notEmpty: {
          msg: "specialSkill must not be empty",
        },
        isIn: {
          args: [["jumping", "sleeping", "purring"]],
          msg: "specialSkill must be either jumping, sleeping, or purring",
        },
      },
    },
    // ...
  }, {});
  Cat.associate = function(models) {
    // associations can be defined here
  };
  return Cat;
};
```

Notice how we **doubly-nest** the list of special skills (`["jumping, "sleeping", "purring"]`) when specifying the `args` for the `isIn` validation. This is because we want to pass **one** argument: an array of three possible special skills.

Now when we try to save a `Cat` with `specialSkill` set to `"pearl diving"`, our code will print:

```
Save failed!
* specialSkill must be either jumping, sleeping, or purring
```

Conclusion

There is a very large variety of validations that are provided by Sequelize. You can find many more in the Sequelize documentation for validations.

Having completed this reading, you now know how to:

1. Validate that an attribute is not set to `NULL`.
2. Validate that a string attribute is not set to the empty string `""`.
3. Validate that a string attribute is not too long (has too many characters).
4. Validate that a numeric attribute meets minimum or maximum thresholds.
5. Validate that an attribute is within a limited set of options.

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:

```
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" FROM "BankA
Executing (default): SELECT "id", "clientName", "balance", "createdAt", "updatedAt" FROM "BankA
Executing (default): UPDATE "BankAccounts" SET "balance"=$1,"updatedAt"=$2 WHERE "id" = $3
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" FROM "BankA
Executing (default): SELECT "id", "clientName", "balance", "createdAt", "updatedAt" FROM "BankA
Executing (208b3951-9ab9-489b-97f0-afb49aaff807): START TRANSACTION;
Executing (208b3951-9ab9-489b-97f0-afb49aaff807): UPDATE "BankAccounts" SET "balance"=$1,"updat
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", "create
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): SELECT "id", "clientName", "balance", "create
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): UPDATE "BankAccounts" SET "balance"=$1,"updat
Executing (76321a03-93c5-47c0-861a-cf24c3e6f3bf): UPDATE "BankAccounts" SET "balance"=$1,"updat
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 With Sequelize Project

In this project, you will build the Data Access Layer to power a Web application. Unlike previously, you will use the Sequelize library and tools to do this to build a more maintainable application.

It has more steps than the SQL version, but it's more maintainable in the long run. Also, the SQL version hid a lot of complexity from you with respect to the running of the SQL. Go look at the SQL version of the files in the **controllers** directory to see what we had to do to load the SQL and execute it.

Now, compare the *simplicity* of those with the simplicity of the files in the **controllers** directory for *this* version of the application. It's easier to understand *this* version. You want to know where to add a column to a table? Go to the migrations. You want to know where to fix a query? Go to the proper repository file.

It's just *so* much better organized.

Quite often, you will see that you will have more files and, overall, more lines of code in well-organized, highly-maintainable software project. Remembering where code is *is hard*. That's why having clearly-named files and directories is so very important.

The data model analysis

This looks no different because it's the same application.

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

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:

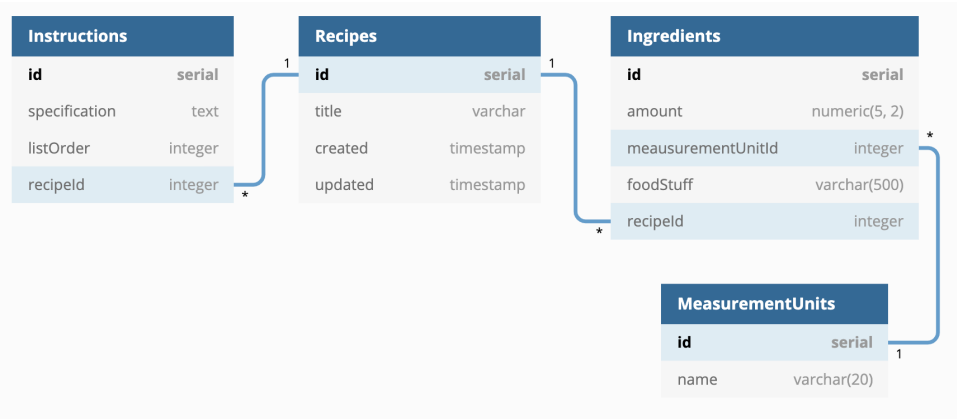
- An amount (optional),
- A unit of measure (optional),
- The actual food stuff, and
- The id of the recipe that it belongs to.

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.



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 already has sequelize and sequelize-cli installed.

Getting started

- Clone the starter project from https://github.com/appacademy-starters/sql-orm-recipe-box
- Run `npm install` to install the packages
- Run `npm run dev` 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 JS files. Each of these will hold your JavaScript code rather than SQL code.

Your code

You're going to be using JavaScript and the tools of Sequelize. Keep the Sequelize documentation open and handy. Even developers that use ORMs every day will keep the documentation open because there's so much to know about them.

Phase 1: Initialize the Sequelize project

Because this project already has sequelize-cli installed, you can initialize the project by typing `npx sequelize-cli init`. The `npx` command runs locally-installed tools. That will create the project structure that Sequelize expects for us to continue to use its tools.

Phase 2: Create a database user for the project

Using a PostgreSQL client like `psql` or Postbird, create a new user for this application named "sequelize_recipe_box_app" with the password "HfKfK79k" *and* the ability to create a database. Here's the link to the CREATE USER documentation so that you can determine which options to give.

Phase 2: Change the connection configuration

The project contains a directory named **config**. Inside there, you will find a file named **config.json**. You need to make some configuration changes.

- Change all the "user" and "password" values to the information for the user that you created in Phase 2.
- Change the "database" values to be "recipe_box_development", "recipe_box_test", and "recipe_box_production".
- Change all of the "dialect" values from "mysql" to "postgres".

- Delete all of the "operatorAliases" entries. It's to support earlier versions of the Sequelize library. Make sure to remove the comma from the preceding line so that it's valid JSON.
- Because you'll be using seed data in this project, add `"seederStorage": "sequelize"` to each of the different blocks so that Sequelize CLI won't run
  a seeder more than once causing duplicate entries in the database.

That will configure the application and the Sequelize tools to properly connect to your development database.

Phase 3: Create your database

Rather than writing SQL to do this, you will use the tools. Run

```
npx sequelize-cli db:create
```

That runs the Sequelize CLI with the command `db:create`.

When you run this, it will default to the "development" setting and read the information from the configuration file to create your database for you! It should print out something like

```
Sequelize CLI [Node: 10.19.0, CLI: 5.5.1, ORM: 5.21.5]

Loaded configuration file "config/config.json".
Using environment "development".
Database recipe_box_development created.
```

You can also drop the database by typing ... you guessed it! The Sequelize CLI with the command `db:drop`!

```
npx sequelize-cli db:drop
```

If you run that, run the "create" command, again, so the database exists.

Phase 4: The units of measurement data

Just as a review, here is the specification for the table that holds units of measurement.

Column Name	Column Type	Constraints
id	SERIAL	PK
name	VARCHAR(20)	NOT NULL

Luckily, the Sequelize models and migrations take care of the "id" property for you without you having to do anything. So, you can just focus on that "name" property.

Create a migration

It's time to create the first migration, the one that defines the table that will hold units of measure. You can use the Sequelize CLI to generate the migration for you. You can *also* tell it to create a model for you, and it will create a migration along *with* the model. You should do that to get the biggest return on investment for the characters that you will type.

The command is `model:generate` and it takes a couple of arguments, "--name" which contains the name of the model (as a singular noun) to generate, and "--attributes" which has a comma-separated list of "property-name:data-type" pairs.

Learning Tip: It is *so very important* that you don't copy and paste this. Type these things out so it has a better chance of creating durable knowledge.

```
npx sequelize-cli model:generate \
  --name MeasurementUnit \
  --attributes name:string
```

That will create two files, if everything works well. (The name of your migration file will be different because it's time-based.)

```
New model was created at models/measurementunit.js
New migration was created at migrations/20200101012349-MeasurementUnit.js
```

The **model** file will be used by the application to query the database. It will be used by the express.js application. It is part of the running software.

The **migration** file is used to construct the database. It is only used by the Sequelize CLI tool to build the database. Unlike those schema and seed files that you had in the SQL version of this project which destroyed *everything* when run, migrations are designed to change your database as your application grows. This is a much better strategy so that existing data in the databases that other people use aren't damaged.

Because the data model requires the "name" column to be both non-null *and* unique, you have to add some information to the migration file. Open it and, for the "name" property, make non-nullable by looking at how the other properties are configured. Then, add the "unique" property set to `true` to the "name" configuration, as well. That should be enough for Sequelize to create the table for you.

The last thing to do is to change the length of the "name" property. By default, Sequelize will make it 255 characters long. The specification for the table says it should really only be 20 characters. To tell the migration that, change the type for "name" from `Sequelize.STRING` to `Sequelize.STRING(20)`.

Run your migration

If you now run your migration with the Sequelize CLI, it will create the table for you.

```
npx sequelize-cli db:migrate
```

That should give you some output that looks similar to this.

```
Loaded configuration file "config/config.json".
Using environment "development".
== 20200101012349-create-measurement-unit: migrating =======
== 20200101012349-create-measurement-unit: migrated (0.021s)
```

You can confirm that the table "MeasurementUnits" is created by using your PostgreSQL client. You'll also see that another table is created, "SequelizeMeta", which contains information about which migration has most recently been run. It contains a single column, "name". Each row contains an entry of which migration file has run. Now that you've run your migration file, the table contains one entry, the name of your migration file. When you run more migrations, you will see more rows, each containing the name of the file that you've run.

psql Note: If you are using `psql` as you PostgreSQL command, be aware that it will lowercase any entity and column names you type in there. If you type `SELECT * FROM MeasurementUnits`, it converts that to `SELECT * FROM measurementunits` before running it. To prevent that from happening, use quotation marks around the table name. `SELECT * FROM "MeasurementUnits"` will do the trick.

It's important that you *never* change the name of a migration file after it's been run.

In the real world, you should *never* change the content of a migration file after it's been committed and shared in your Git repository. Asking others to

rollback their migrations just because you changed one of yours is bad manners. Instead, you should add a new migration that makes the change that you want.

Create the seed data

You can create the seed data for the unit of measurements by creating a **seeder** as the Sequelize CLI calls them. You can create one using the Sequelize CLI tool. Run the following and make sure you don't get any errors.

```
npx sequelize-cli seed:generate --name default-measurement-units
```

Now, you want to insert the seed data. You will do this by using the `bulkInsert` method of the object passed in through the `queryInterface` parameter of the `up` method. Feel free to delete the comment in the `up` method and replace it with this.

```
return queryInterface.bulkInsert('MeasurementUnits', [
  { name: 'cups', createdAt: new Date(), updatedAt: new Date() },
]);
```

The `bulkInsert` method takes two parameters:

- The name of the table to insert into, and
- An array of objects that have property names that match the column names in the table.

You can see that the first object has been provided by the example. Now, create objects for all of these values, as well. (The empty item in the list is an empty string and is intentional) Make sure you do them **in this order,** or when we get to the seed data for the other tables it won't work. (We've supplied you with files for the seed data for the other tables because there is a lot of it)

- "fluid ounces"
- "gallons"
- "grams"
- "liters"
- "milliliters"
- "ounces"
- "pinch"
- "pints"
- "pounds"
- "quarts"
- "tablespoons"
- "teaspoons"
- ""
- "cans"
- "slices"
- "splash"

Now, run the Sequelize CLI with the command `db:seed:all`.

After you get that done, you can confirm that all of the records (rows) were created in the "MeasurementUnits" table.

Phase 5: The recipe table model

This will go much like the last one, except there's no seed data. Just to refresh your memory, here's the specification for the "recipes" table.

Column Name	Column Type	Constraints
id	SERIAL	PK
title	VARCHAR(200)	NOT NULL

Column Name	Column Type	Constraints
created	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP
updated	TIMESTAMP	NOT NULL, DEFAULT CURRENT_TIMESTAMP

As you've discovered, Sequelize takes care of the "id" for you *and* the columns to track when the recipe has been created and updated! Your job is to

- Generate a model for the "recipe"
- Customize the migration so the "title" column is not nullable

Run your migration and confirm that you defined it correctly by checking the attributes in the description of the table. The important parts to check are that the "title" column is a VARCHAR(200) and is non-nullable. (The "Collation" column has been removed for brevity.)

```
                Table "public.Recipes"
  Column   |           Type           | Nullable |  Default
-----------+--------------------------+----------+-----------
 id        | integer                  | not null | nextval(...
 title     | character varying(200)   | not null |
 createdAt | timestamp with time zone | not null |
 updatedAt | timestamp with time zone | not null |
Indexes:
    "Recipes_pkey" PRIMARY KEY, btree (id)
```

Phase 6: The instruction table model

Now, things get a little trickier because this model will reference the recipe model. Here's the specification for the "instructions" table.

Column Name	Column Type	Constraints
id	SERIAL	PK
specification	TEXT	NOT NULL
listOrder	INTEGER	NOT NULL
recipeId	INTEGER	FK, NOT NULL

When you type out your migration generation command, the "--attributes" parameter will look like this:

```
--attributes column1:type1,column2:type2,column3:type3
```

Instead of using "string" for the "specification" column of the table, use "text" to generate a TEXT column.

After it generates the migration file, modify each of the column descriptors in the migration so that the columns are not nullable. Then, add a new property to the one for "recipeId" called "references" that is an object that contains a "model" property set to "Recipes". It should look like this.

```
recipeId: {
  allowNull: false,
  references: { model: "Recipes" },
  type: Sequelize.INTEGER,
},
```

With that in place, run the migration. Then, check the table definition in your PostgreSQL client.

```
                Table "public.Instructions"
    Column     |           Type           | Nullable |     Default
---------------+--------------------------+----------+-----------------
 id            | integer                  | not null | nextval('"Ins...
 specification | text                     | not null |
 listOrder     | integer                  | not null |
 recipeId      | integer                  | not null |
 createdAt     | timestamp with time zone | not null |
 updatedAt     | timestamp with time zone | not null |
Indexes:
    "Instructions_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "Instructions_recipeId_fkey" FOREIGN KEY ("recipeId")
                                 REFERENCES "Recipes"(id)
```

You should see all non-null columns and a foreign key between the "Instructions" table and the "Recipes" table.

Phase 7: The ingredients model

The model for ingredients has *two* foreign keys. Create the model and migration for it. Here's the table specification.

Column Name	Column Type	Constraints
id	SERIAL	PK
amount	NUMERIC(5, 2)	NOT NULL
measurementUnitId	INTEGER	FK, NOT NULL
foodStuff	VARCHAR(500)	NOT NULL
recipeId	INTEGER	FK, NOT NULL

After you modify and run your migration, you should have a table in your database that looks like this, with two foreign keys, one to the "Recipes" table and the other to the "MeasurementUnits" table.

```
                      Table "public.Ingredients"
      Column       |           Type           | Nullable |      Default
-------------------+--------------------------+----------+-----------------
 id                | integer                  | not null | nextval('"Ing...
 amount            | numeric(5,2)             | not null |
 measurementUnitId | integer                  | not null |
 foodStuff         | character varying(500)   | not null |
 recipeId          | integer                  | not null |
 createdAt         | timestamp with time zone | not null |
 updatedAt         | timestamp with time zone | not null |
Indexes:
    "Ingredients_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "Ingredients_measurementUnitId_fkey"
        FOREIGN KEY ("measurementUnitId")
        REFERENCES "MeasurementUnits"(id)
    "Ingredients_recipeId_fkey"
        FOREIGN KEY ("recipeId")
        REFERENCES "Recipes"(id)
```

Phase 8: Seed data for all of the tables

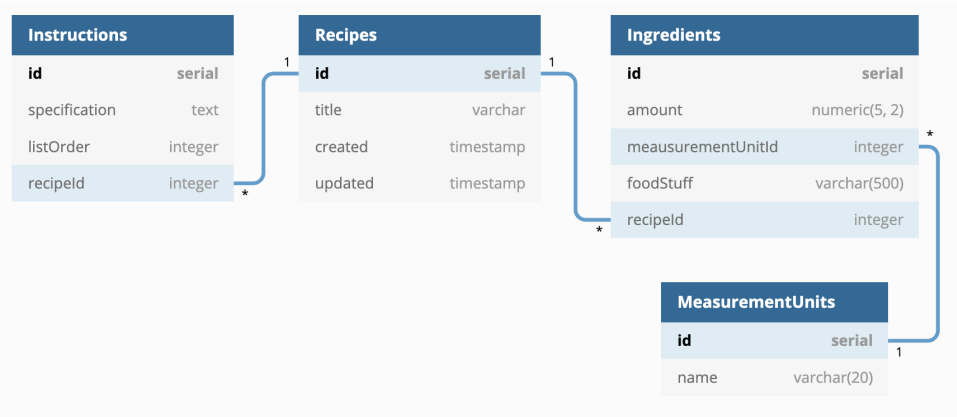
Now that you have tables in the database, it's time to create some seed data for all of them. In the **data-access-layer** directory, you will find three text files each containing JavaScript objects on each row that match the tables in the previous three sections.

If you didn't seed the MeasurementUnits data in the correct order listed in the section above, you may have to redo that seed file, because the data from the text files depends on the ids of the data in the `MeasurementUnits` table being correct.

There are three tables to seed: Ingredients, Instructions, and Recipes. It is important to note that you will need to seed them in the correct order due to

foreign key dependencies.

Look at the data model for the application, again.



You can see that the Instructions depends on Recipes because it has the foreign key "recipeId" to the Recipes table. You can also see that the Ingredients table has dependencies on the Recipes and MeasurementUnits tables because of its foreign keys "measurementUnitId" and "recipeId". (You've already seeded the MeasurementUnits table in Phase 4, so that data exists for use by the Ingredients table.) Recipes does not have any foreign keys. You need to seed Recipes, first, because it does not have any foreign keys and, therefore, does not have any data dependencies. Then, you can seed the Instructions and Ingredients tables in either order because their data dependencies will have been met.

Create seeder files for them in that order: Recipes, first, then Ingredients and Instructions. Use the contents of each of the text files in **data-access-layer** to do bulk inserts.

After you create each seed file, run

```
npx sequelize-cli db:seed:all
```

to make sure you don't have any errors. If you do, fix them before moving onto the next seed file.

If you end up seeding the data in the wrong order and getting a foreign key constraint error, just use the CLI to drop the database, create the database, migrate the database, and then you can try running your seeders, again. You may need to rename your migration filenames to get your seeds running in the correct order.

Phase 9: Updating models with references

Now that you have all of the migrations set up correctly and a database defined, it is time for you to turn your attention to the model files that were generated in the previous phases.

Consider the relationship between an Instruction and a Recipe. A Recipe *has many* Instructions. In the other direction, you would say that an Instruction *has one* Recipe, or that Instruction *belongs to* the Recipe. To set that up in your model, open the file **models/recipe.js**. In there, you will see the following.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Recipe = sequelize.define('Recipe', {
    title: DataTypes.STRING
  }, {});
  Recipe.associate = function(models) {
    // associations can be defined here
  };
  return Recipe;
};
```

In the `associate` function is where you can define the association between the Recipe and the Instruction. Replace the comment with the following statement.

```
Recipe.hasMany(models.Instruction, { foreignKey: 'recipeId' });
```

This instructs Sequelize that Recipe should have a collection of Instruction objects associated with it. To insure that Sequelize uses the foreign key column that you created on the "Instructions" table in your migration, you must specify it as part of the collection definition.

In the file **models/instruction.js**, replace the comment with the following to define the other side of the relationship.

```
Instruction.belongsTo(models.Recipe, { foreignKey: 'recipeId' });
```

This instructs Sequelize that Instruction has a single Recipe object associated with it. Again, because of inconsistent naming conventions used by Sequelize, you must specify the foreign key column name in the "Instructions" table.

Think about the many-to-one and one-to-many relationships between Ingredient, MeasurementUnit, and Recipe. Then, modify those model files accordingly with the `hasMany` and `belongsTo` associations, always specifying the name of the foreign key column that binds the two tables together.

Phase 10: Updating models with validations

Now that you have seed data created, it will be important to prevent users from entering data that does not meet the expectations of the data model.

Consider the content of **models/instruction.js**

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Instruction = sequelize.define('Instruction', {
    specification: DataTypes.TEXT,
    listOrder: DataTypes.INTEGER,
    recipeId: DataTypes.INTEGER
  }, {});
  Instruction.associate = function(models) {
    Instruction.belongsTo(models.Recipe, { foreignKey: 'recipeId' });
  };
  return Instruction;
};
```

It would be nice if the model could validate each of those properties to make sure that no one sets them to null and that `listOrder` is greater than 0, for example. You can do that with per-attribute validations.

For example, you can change the above code to the following to make sure that the "specification" property won't get set to an empty string when someone tries to save the object.

```
'use strict';
module.exports = (sequelize, DataTypes) => {
  const Instruction = sequelize.define('Instruction', {
    specification: {
      type: DataTypes.TEXT,
      validate: {
        notEmpty: true,
      },
    },
    listOrder: DataTypes.INTEGER,
    recipeId: DataTypes.INTEGER
  }, {});
  Instruction.associate = function(models) {
    Instruction.belongsTo(models.Recipe, { foreignKey: 'recipeId' });
  };
  return Instruction;
};
```

Make sure all of the other string properties in the models won't allow the empty string to be set on them.

Phase 11: Cascade delete for recipes

The Recipe model has dependencies: the Instruction and the Ingredient both have *belongs to* relationships. This means that the row in the "Recipes" table must exist to have records in the "Ingredients" and "Instructions" table. If you try to delete a Recipe row from the database that has either Instructions or Ingredients, it won't work due to referential integrity. You would have to delete all of the Ingredients and Instructions *before* being able to delete the Recipe.

Sequelize provides a handy shortcut for that and will manage deleting the associated records for you when you delete a row from the Recipes table. It's called a *cascading delete*. Open the **models/recipe.js** file. In there, modify the second argument of each of the `hasMany` calls to include two new property/value pairs:

- `onDelete: 'CASCADE'`
- `hooks: true`

Refer to the documentation on Associations to see an example. But, don't delete the `foreignKey` property that you put there in Phase 9.

Phase 12: Building the repositories

Now that you have the seeds, models, and migrations out of the way, you can build the data access layer with a lot of speed. Sequelize will now handle all of the SQL generation for you. You can just use the models that you've painstakingly crafted.

Because you are writing JavaScript files, you want the server to restart because it won't automatically reload the changed JavaScript that you're writing. To that end, you will use a different command while developing.

```
npm run dev
```

This runs a special script that will reload the JavaScript in the data access layer every time you make a change. You can see what's run in the **package.json** file in this project in the "scripts" section for the "dev" property.

You will work in the three files named

- **recipes-repository.js**: The collection of functions needed to interact with recipes for the application
- **instructions-repository.js**: The collection of functions needed to interact with the instructions for the application
- **ingredients-repository.js**: The collection of functions needed to interact with the ingredients for the application

Each of the files imports your models and makes them available to you. Then, you can use them in your querying. Follow the hints in each of the repository functions.