WEEK-10 DAY-1
Hello, Database!

Make It Pretty Learning Objectives

Portfolio Quality

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

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

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

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

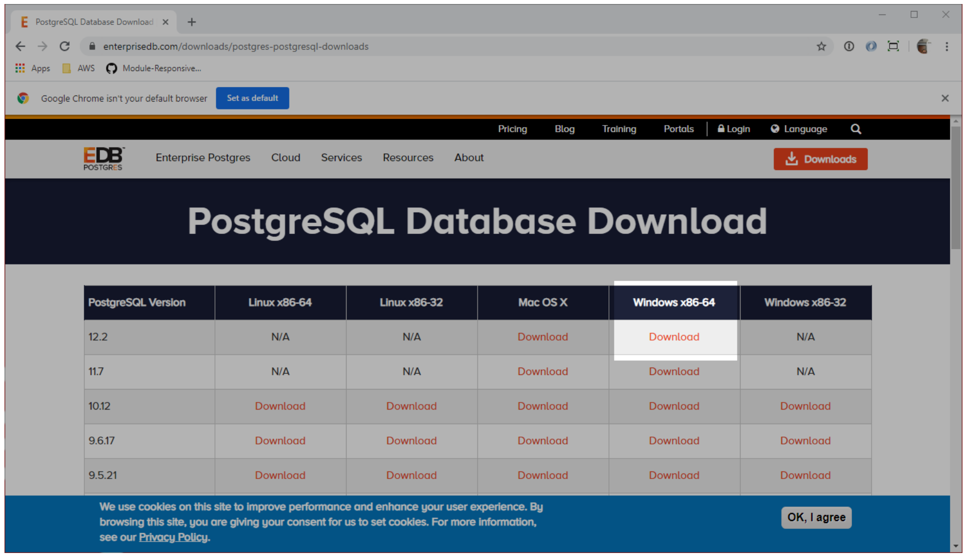
You will install three pieces of software so that you can start using PostgreSQL. You will install PostgreSQL itself on your Windows installation. Then, you will install `psql` in your Ubuntu installation. Then you will also install Postbird, a cross-platform graphical user interface that makes working 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.



PostgreSQL Database Download

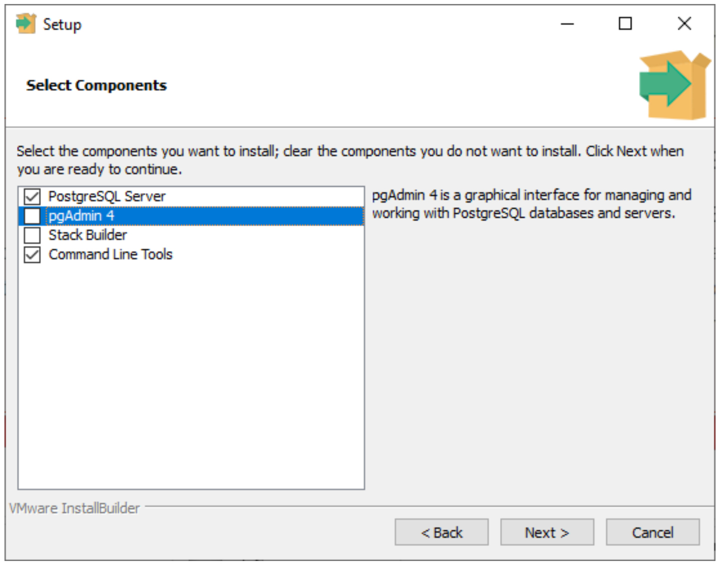
PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
12.2	N/A	N/A	Download	Download	N/A
11.7	N/A	N/A	Download	Download	N/A
10.12	Download	Download	Download	Download	Download
9.6.17	Download	Download	Download	Download	Download
9.5.21	Download	Download	Download	Download	Download

We use cookies on this site to improve performance and enhance your user experience. By browsing this site, you are giving your consent for us to set cookies. For more information, see our Privacy Policy.

OK, I agree

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.



Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- [✓] PostgreSQL Server
- [ ] pgAdmin 4
- [ ] Stack Builder
- [✓] Command Line Tools

pgAdmin 4 is a graphical interface for managing and working with PostgreSQL databases and servers.

- 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-key add -
```

If prompted for your password, type it.

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

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".

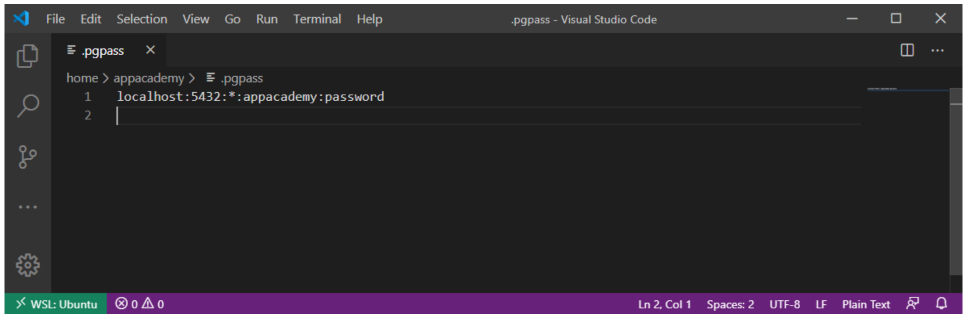
Now you will create your PostgreSQL password file. Type the following into your Ubuntu shell to open Visual Studio Code and create a new file.

```
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.


File Edit Selection View Go Run Terminal Help
.pgpass - Visual Studio Code
.pgpass
home > appacademy > .pgpass
WSL: Ubuntu
Ln 2, Col 1 Spaces: 2 UTF-8 LF Plain Text

```
localhost:5432:*:appacademy:password
```

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
```

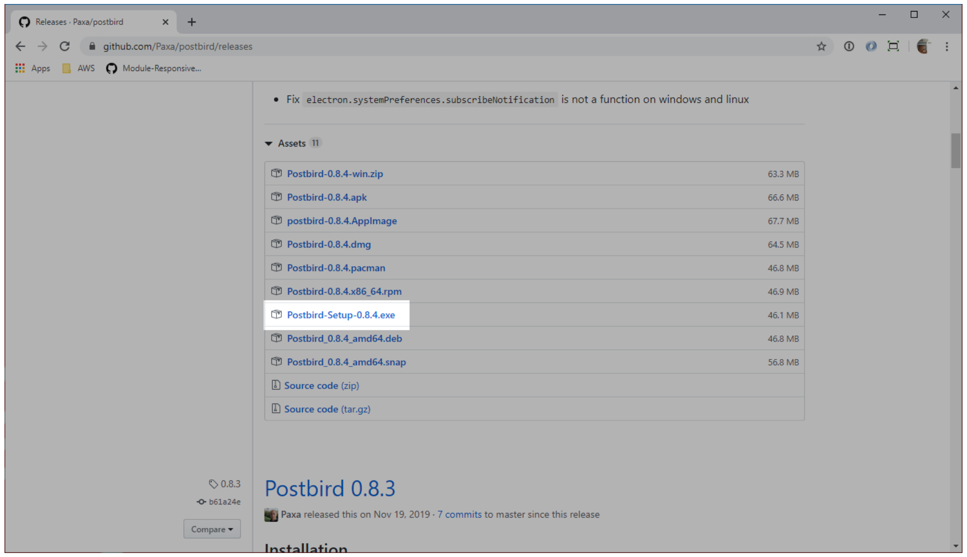
You can confirm that only you have read/write permission by typing `ls -al ~/.pgpass`. That should return output that looks like this, **with your Ubuntu user name instead of "appacademy".**

```
-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".



- Fix `electron.systemPreferences.subscribeNotification` is not a function on windows and linux

Assets 11

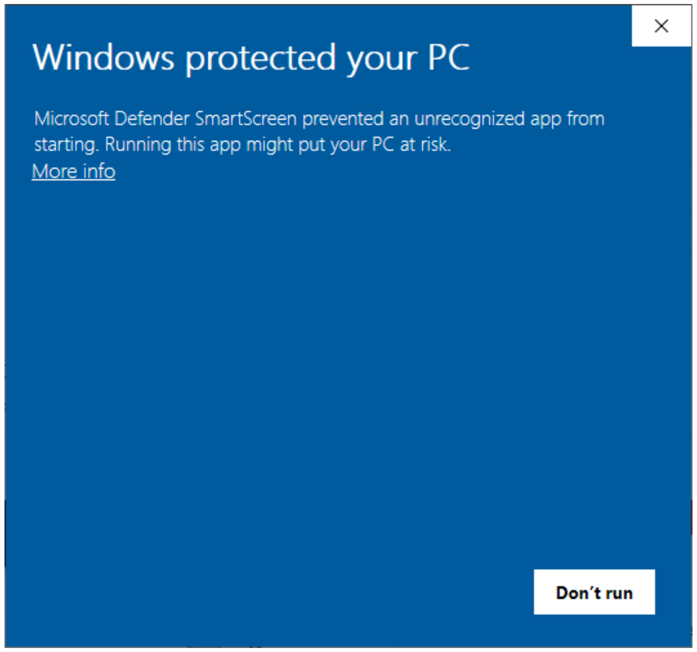
Postbird-0.8.4-win.zip	63.3 MB
Postbird-0.8.4.apk	66.6 MB
postbird-0.8.4.AppImage	67.7 MB
Postbird-0.8.4.dmg	64.5 MB
Postbird-0.8.4.pacman	46.8 MB
Postbird-0.8.4.x86_64.rpm	46.9 MB
Postbird-Setup-0.8.4.exe	46.1 MB
Postbird_0.8.4_amd64.deb	46.8 MB
Postbird_0.8.4_amd64.snap	56.8 MB
Source code (zip)	
Source code (tar.gz)	

Postbird 0.8.3

Paxa released this on Nov 19, 2019 · 7 commits to master since this release

Installation

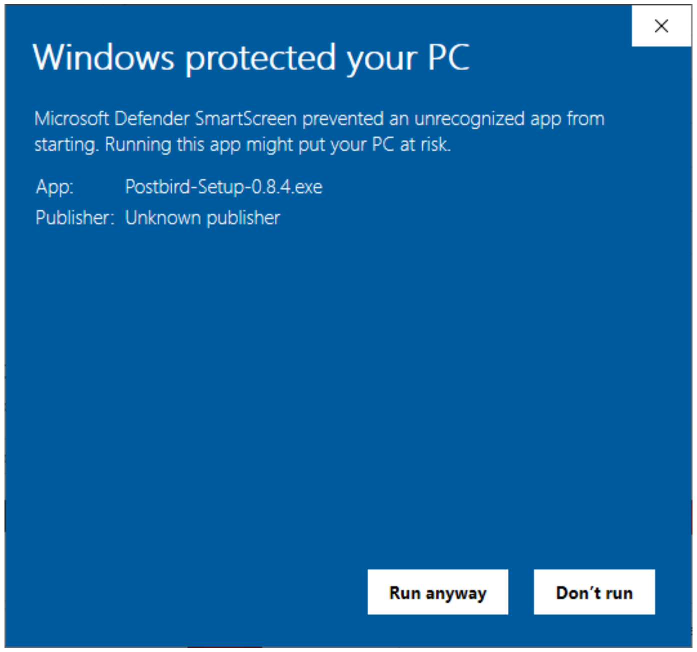
After that installer downloads, run it. You will get a warning from Windows that this is from an unidentified developer. If you don't want to install this, find a PostgreSQL GUI client that you do trust and install it or do everything from the command line.


Windows protected your PC

Microsoft Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.
More info

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.


Windows protected your PC

Microsoft Defender SmartScreen prevented an unrecognized app from starting. Running this app might put your PC at risk.

App: Postbird-Setup-0.8.4.exe
Publisher: Unknown publisher

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.

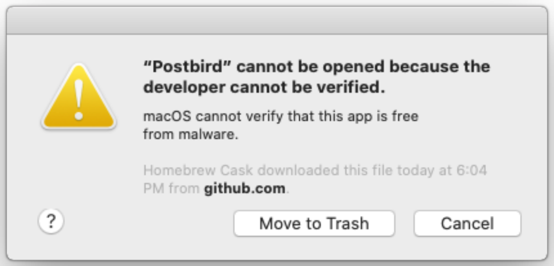
```
==> Casks
postbird
```

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.

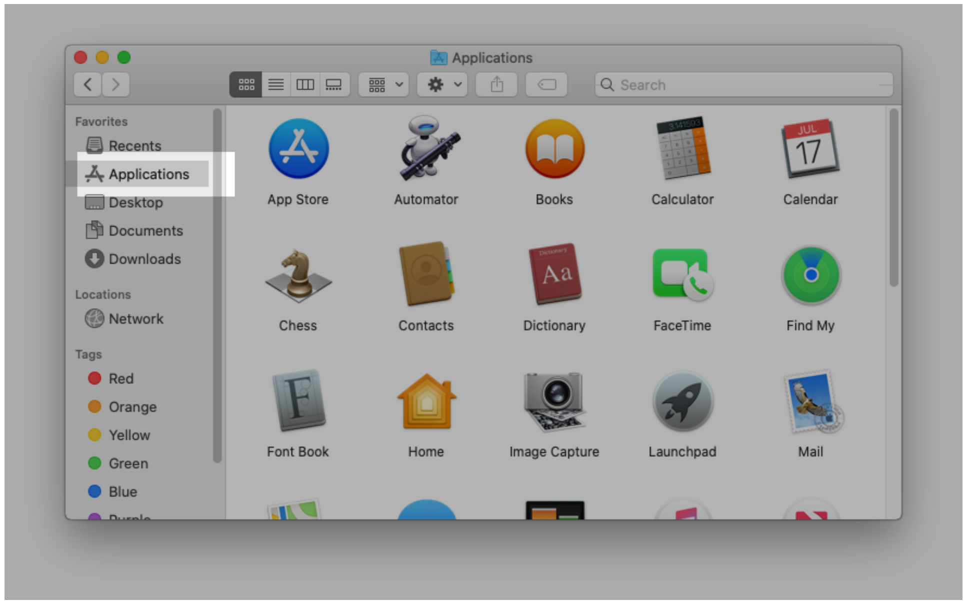

"Postbird" cannot be opened because the developer cannot be verified.

macOS cannot verify that this app is free from malware.

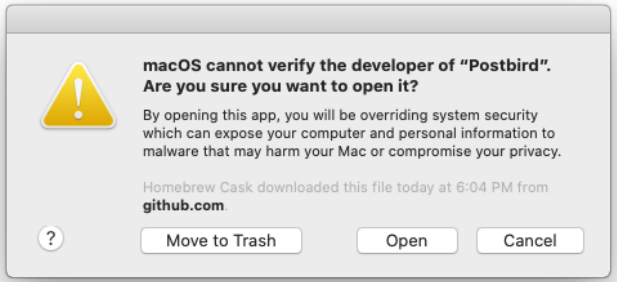
Homebrew Cask downloaded this file today at 6:04 PM from github.com.

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.


macOS cannot verify the developer of “Postbird”. Are you sure you want to open it?

By opening this app, you will be overriding system security which can expose your computer and personal information to malware that may harm your Mac or compromise your privacy.

Homebrew Cask downloaded this file today at 6:04 PM from **github.com**.

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 privileges
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=CTc/appacademy
template1	appacademy	UTF8	C	C	=c/appacademy +
					appacademy=CTc/appacademy

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 - 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.

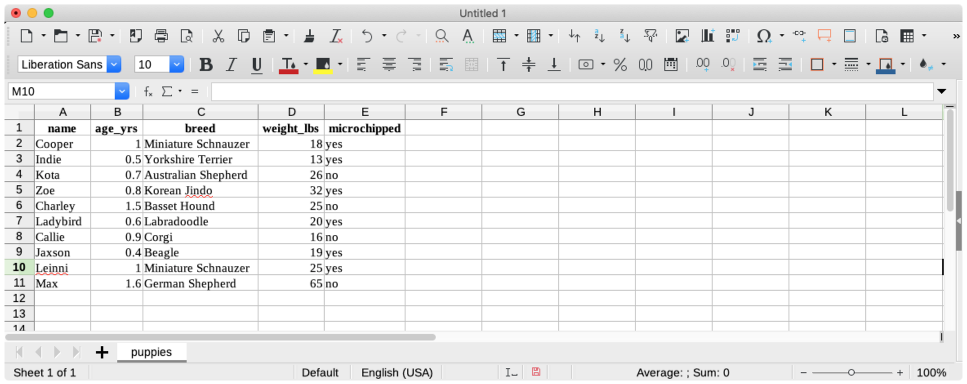
You can now create users that can connect to the relational database management system. You can now create databases and secure them so only certain users can connect to them. Now, it's time to get into the part where you define the entities that actually hold the data: **tables**!

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.



	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

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.

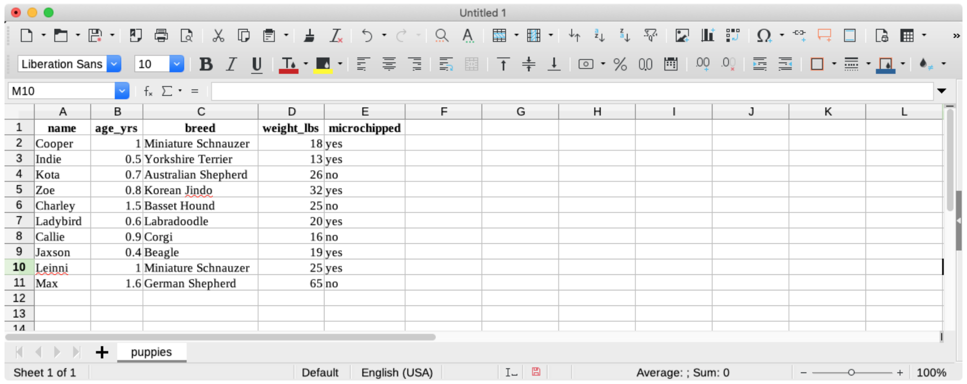
Table Management - Part II

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.



	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

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.

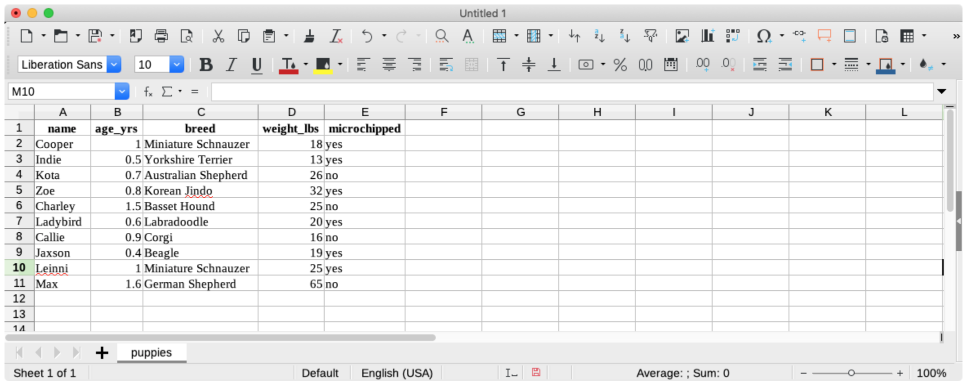
For the "puppies" table, a reasonable default value for the "microchipped" column would be `FALSE`. You can add that to your SQL using the `DEFAULT` keyword.

```
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.



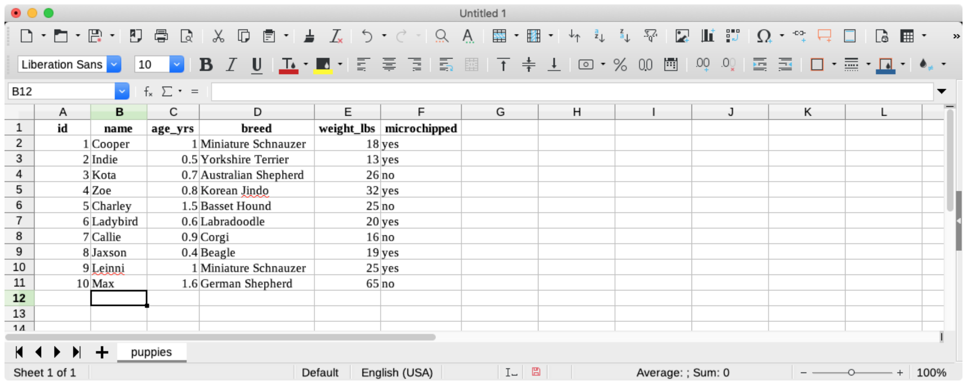
	A	B	C	D	E
1	name	age_yrs	breed	weight_lbs	microchipped
2	Cooper	1	Miniature Schnauzer	18	yes
3	Indie	0.5	Yorkshire Terrier	13	yes
4	Kota	0.7	Australian Shepherd	26	no
5	Zoe	0.8	Korean Jindo	32	yes
6	Charley	1.5	Basset Hound	25	no
7	Ladybird	0.6	Labradoodle	20	yes
8	Callie	0.9	Corgi	16	no
9	Jaxson	0.4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

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.

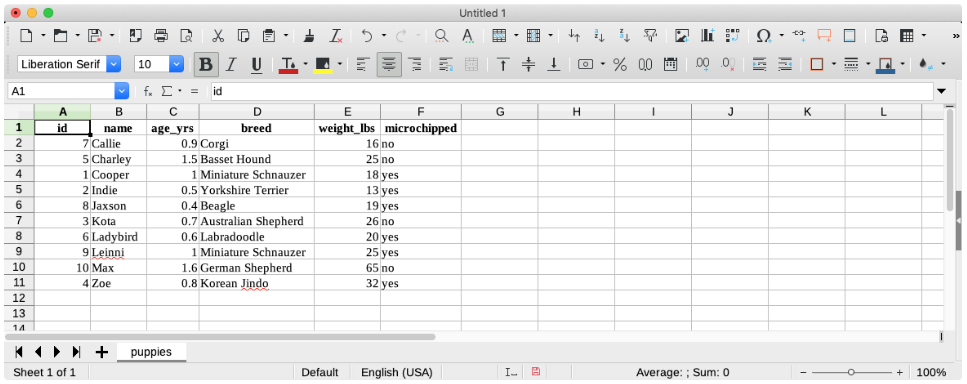
Now, what happens when you are tracking 300 dogs in the spreadsheet? What happens when your spreadsheet has 17 dogs named "Max"? It is helpful to have some way to uniquely identify a row in the spreadsheet. This is the idea behind a **primary key**. You can specify a column to be the primary key with 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
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

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.



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

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'::regclass)
 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)
```

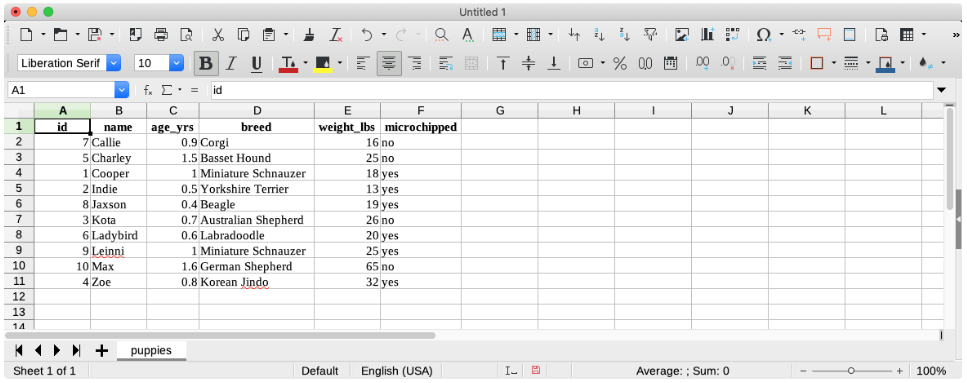
Down there at the bottom, you see that PostgreSQL has added a `UNIQUE CONSTRAINT` to the list of indexes for the "email" field. Now, if someone tried to put an email address into the table that someone had previously used, then the database would return an error.

```
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.

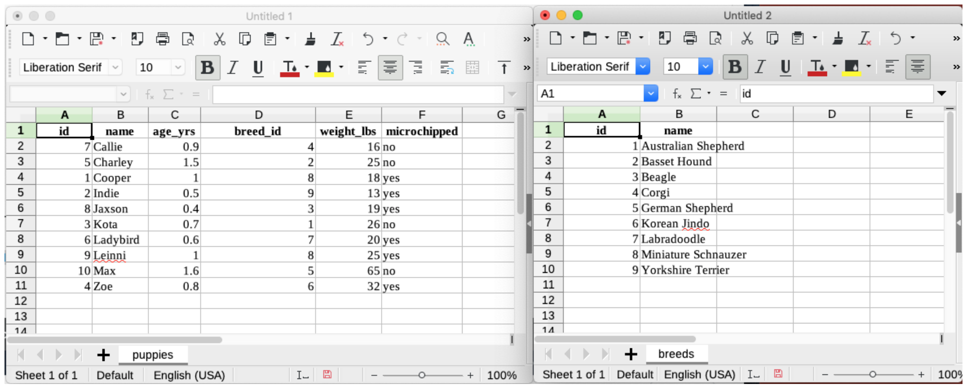
This process is called **normalization**. It's a *really big deal* in database communities. And, it's a really big deal for application developers to maintain the integrity of the data. Bad data leads to bad applications.

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.



id	name	age_yrs	breed_id	weight_lbs	microchipped
7	Callie	0.9	4	16	no
5	Charley	1.5	2	25	no
1	Cooper	1	8	18	yes
2	Indie	0.5	9	13	yes
8	Jaxson	0.4	3	19	yes
3	Kota	0.7	1	26	no
6	Ladybird	0.6	7	20	yes
9	Leinni	1	8	25	yes
10	Max	1.6	5	65	no
4	Zoe	0.8	6	32	yes

id	name
1	Australian Shepherd
2	Basset Hound
3	Beagle
4	Corgi
5	German Shepherd
6	Korean Jindo
7	Labradoodle
8	Miniature Schnauzer
9	Yorkshire Terrier

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.

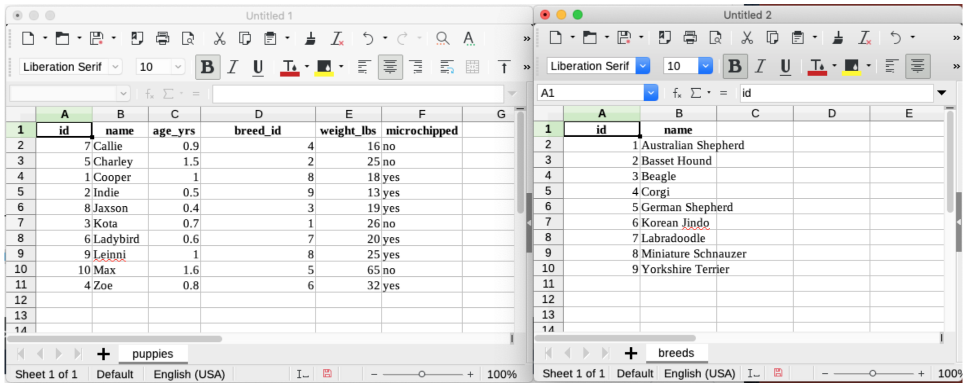
Here's what the new "puppies" SQL looks like. Drop the old "puppies" table and type this SQL in there.

```
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.



id	name	age_yrs	breed_id	weight_lbs	microchipped
7	Callie	0.9	4	16	no
5	Charley	1.5	2	25	no
1	Cooper	1	8	18	yes
2	Indie	0.5	9	13	yes
8	Jaxson	0.4	3	19	yes
3	Kota	0.7	1	26	no
6	Ladybird	0.6	7	20	yes
9	Leinni	1	8	25	yes
10	Max	1.6	5	65	no
4	Zoe	0.8	6	32	yes

id	name
1	Australian Shepherd
2	Basset Hound
3	Beagle
4	Corgi
5	German Shepherd
6	Korean Jindo
7	Labradoodle
8	Miniature Schnauzer
9	Yorkshire Terrier

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

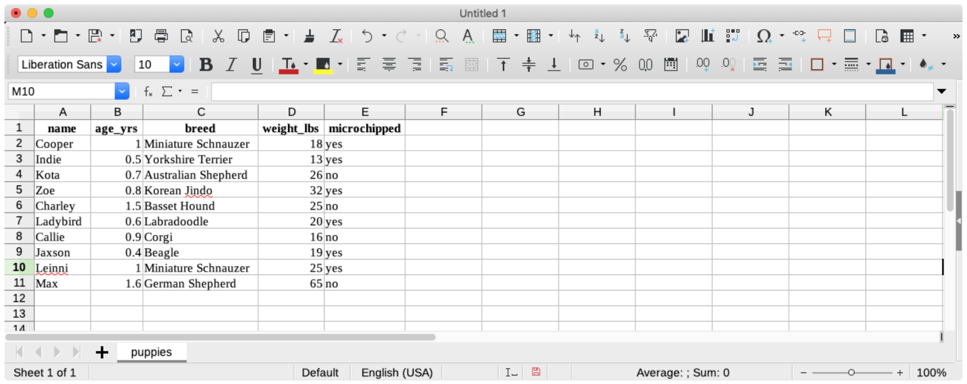
Puppies and breed spreadsheets normalized: images/spreadsheet-puppies-and-breeds-normalized.pngimages/spreadsheet-puppies-and-breeds-normalized.pngort "[TOC]" {cmd="toc" depthFrom=2 depthTo=6 orderedList=false} -->

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
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

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.

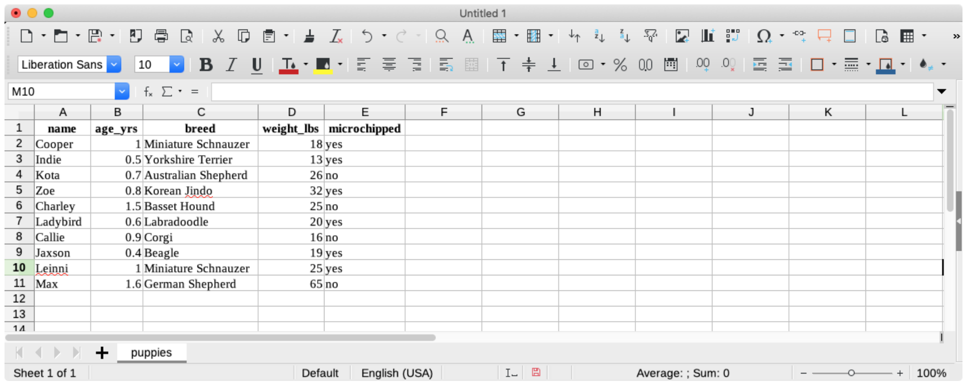
For the "puppies" table, a reasonable default value for the "microchipped" column would be `FALSE`. You can add that to your SQL using the `DEFAULT` keyword.

```
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.

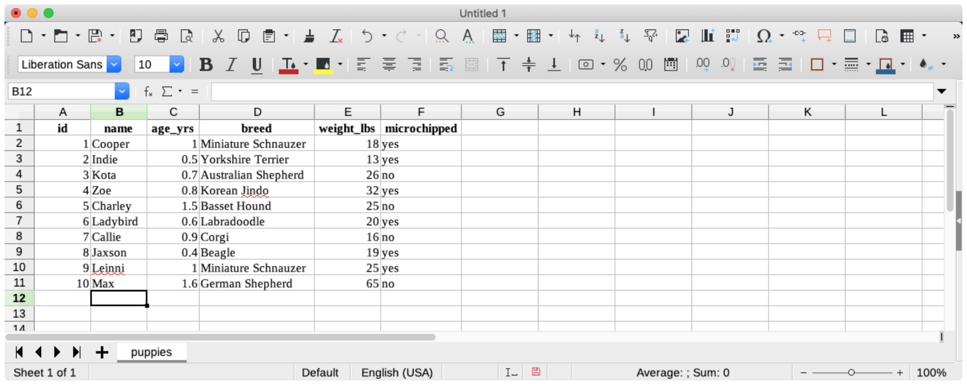


name	age_yrs	breed	weight_lbs	microchipped
Cooper	1	Miniature Schnauzer	18	yes
Indie	0.5	Yorkshire Terrier	13	yes
Kota	0.7	Australian Shepherd	26	no
Zoe	0.8	Korean Jindo	32	yes
Charley	1.5	Basset Hound	25	no
Ladybird	0.6	Labradoodle	20	yes
Callie	0.9	Corgi	16	no
Jaxson	0.4	Beagle	19	yes
Leinni	1	Miniature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

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.

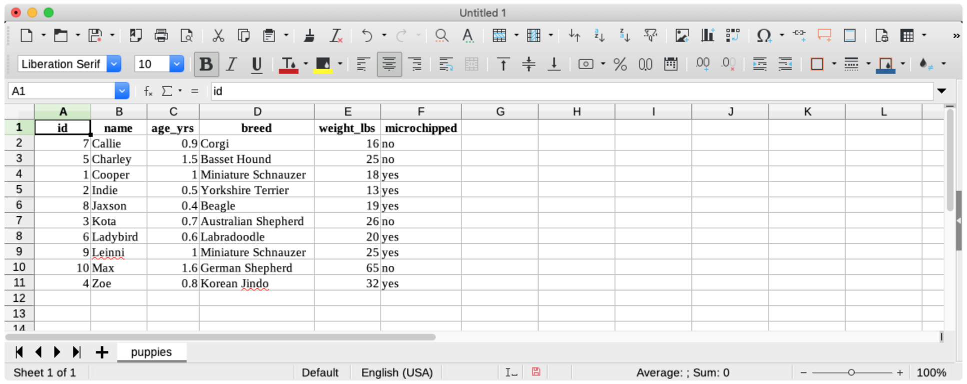
Now, what happens when you are tracking 300 dogs in the spreadsheet? What happens when your spreadsheet has 17 dogs named "Max"? It is helpful to have some way to uniquely identify a row in the spreadsheet. This is the idea behind a **primary key**. You can specify a column to be the primary key with 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.



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

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.



	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

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'::regclass)
 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)
```

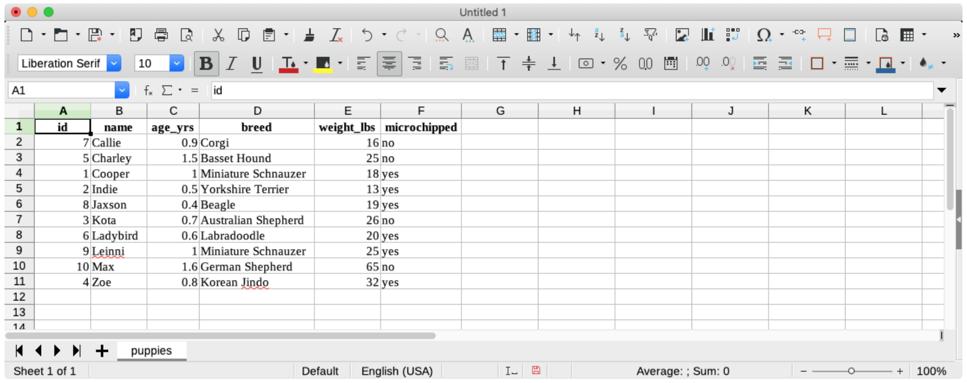
Down there at the bottom, you see that PostgreSQL has added a `UNIQUE CONSTRAINT` to the list of indexes for the "email" field. Now, if someone tried to put an email address into the table that someone had previously used, then the database would return an error.

```
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.

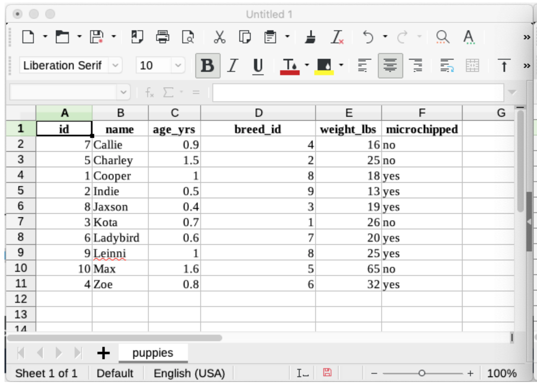
This process is called **normalization**. It's a *really big deal* in database communities. And, it's a really big deal for application developers to maintain the integrity of the data. Bad data leads to bad applications.

To do this follows a fairly simple set of steps.

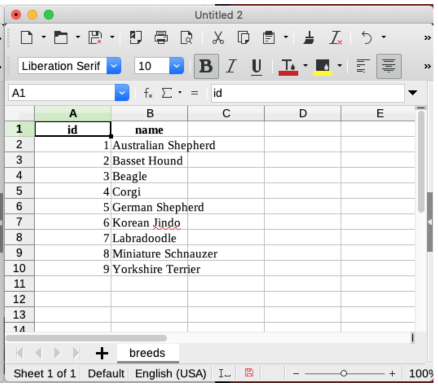
1. Figure out what related data repeats itself. In this case, it is only the single column that contains the **breed** names.
2. Create a new table to hold that data. Make sure it has a primary key. In this case, you can create a "breeds" table that contains an "id" the name of the breed.
3. Replace 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.



id	name	age_yrs	breed_id	weight_lbs	microchipped
7	Callie	0.9	4	16	no
5	Charley	1.5	2	25	no
1	Cooper	1	8	18	yes
2	Indie	0.5	9	13	yes
8	Jaxson	0.4	3	19	yes
3	Kota	0.7	1	26	no
6	Ladybird	0.6	7	20	yes
9	Leinni	1	8	25	yes
10	Max	1.6	5	65	no
4	Zoe	0.8	6	32	yes



id	name
1	Australian Shepherd
2	Basset Hound
3	Beagle
4	Corgi
5	German Shepherd
6	Korean Jindo
7	Labradoodle
8	Miniature Schnauzer
9	Yorkshire Terrier

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.

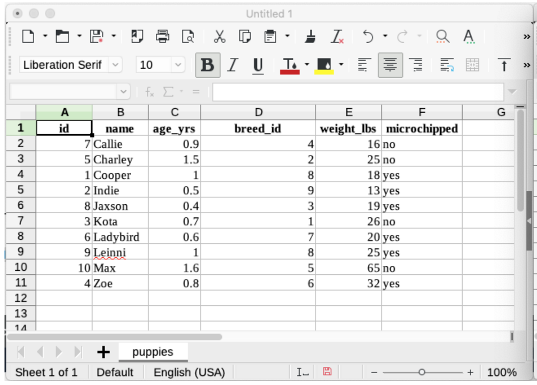
Here's what the new "puppies" SQL looks like. Drop the old "puppies" table and type this SQL in there.

```
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)
);
```

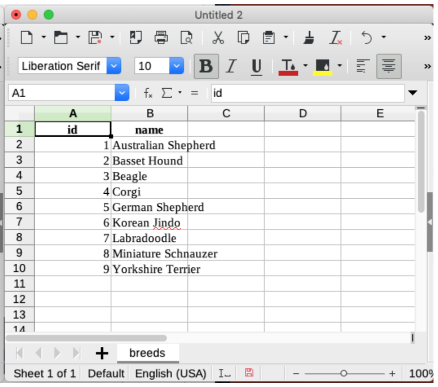
That new thing at the bottom of the definition, that's how you relate one table to another. If follows the syntax

```
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.



id	name	age_yrs	breed_id	weight_lbs	microchipped
7	Callie	0.9	4	16	no
5	Charley	1.5	2	25	no
1	Cooper	1	8	18	yes
2	Indie	0.5	9	13	yes
8	Jaxson	0.4	3	19	yes
3	Kota	0.7	1	26	no
6	Ladybird	0.6	7	20	yes
9	Leinni	1	8	25	yes
10	Max	1.6	5	65	no
4	Zoe	0.8	6	32	yes



id	name
1	Australian Shepherd
2	Basset Hound
3	Beagle
4	Corgi
5	German Shepherd
6	Korean Jindo
7	Labradoodle
8	Miniature Schnauzer
9	Yorkshire Terrier

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

---

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