# Learn Bash by Building a Boilerplate

**The first thing you need to do is start the terminal.** Do that by clicking the "hamburger" menu at the top left of the screen, going to the "terminal" section, and clicking "new terminal". Once you open a new one, type echo hello terminal into the terminal and press enter.

What you see in the terminal below is a folder (or directory) on this machine. Type pwd into the terminal and press enter to see the path of the folder. pwd stands for "print working directory".

The output tells you where the folder you are in is located. You are in the project folder, which is in the CodeAlly folder, which is in the home folder. Type ls into the terminal to see what's in this folder. ls stands for "list".

The output is showing everything in this folder. There's two folders here. You can use cd <folder\_name> to go into a folder. cd stands for "change directory". Change to the freeCodeCamp directory.

You are in the freecodecamp folder now. You may have noticed that the prompt changed to include it. Print the working directory of the freeCodeCamp folder to see the full path of where you are.

You can see the path of the freeCodeCamp folder. It's in the project folder you were just in. List the contents of the freeCodeCamp folder to see what's here.

There's two folders and three files here. The folders are blue and the files include their extension. Next, change to that test directory.

You can see you are in the test folder now. It shows test in the prompt. Print the full path of this directory. Remember that "folder" and "directory" are the same thing.

That's the path to the test folder, it's in the freeCodeCamp folder. **List** the contents of this folder.

These are all files. There's no more folders to go into here. You can use cd .. to go back a folder level. The two dots will take you back one level. Go back to the freeCodeCamp folder.

test got removed from the prompt since you left that folder and you're back in the freeCodeCamp folder. List the contents of what's here to remind yourself.

There's the test folder you were just in. You can see what's in a file with more <filename>. Use it to view what's in the package.json file.

It looks like a JSON object. You can empty the terminal with clear. The terminal looks a little cluttered, why don't you clear it.

Now you have a fresh screen 😄 List what's in here again.

You checked out the test folder and the package.json file. What next? Why don't you go into that node\_modules directory.

Now the prompt includes node\_modules since that's where you are. List what's in the here.

That's a lot of folders. You can add a **flag** to a command to use it different ways like this: ls <flag>. List the contents of the node\_modules folder in "long list format". Do that by adding the -l flag to the "list" command.

It is showing more details about each item in here and it's a little easier to read. One of the folders is named has, why don't you change into it.

You are now in the has folder. List its contents.

There's a few files and folders here. Can you tell the difference? Take a look at **more** of that README.md file.

Nothing noteworthy in there. You can't see what's in the here anymore, list the contents again.

That one file doesn't appear to have an extension. Strange. Take a look at **more** of the that "license" file that doesn't show an extension.

Pretend you read all that. It looks a little messy in here again so why don't you clear the terminal.

Better. Remind yourself what's in here with the list command.

Go into that src directory to see what you can find in there.

View the full path of this folder.

Getting deeper still. You can see that each new folder has a / in front of it. Take a look at what's in this folder.

Only one file here. Show me what's in it with more.

It's some JavaScript 😄 I think you've fooled around enough. Why don't you navigate out of here. Change back to the has directory.

You're getting pretty good. Change back to the node\_modules directory.

You can go back two folders with cd ../... Each set of dots represents another folder level. Go back to the project directory from the node\_modules directory.

You are back in the project folder where you started. List what's in here again.

That's right. Why don't you get a fresh start by clearing the terminal.

You will be making a website boilerplate. You can make a new folder with mkdir <folder\_name>. mkdir stands for "make directory". Make a website directory in this project folder. Remember that "directory" and "folder" mean the same thing.

List what's here to make sure it got created.

It worked. The website files will be in the new directory. Change to the website directory so you can start creating them.

List the contents of the website folder.

It's brand new, there's nothing in it yet. The echo command lets you print anything to the terminal. You used it in the first lesson. Just type what you want to print after it. Use it to print hello website to the terminal.

Websites usually have an index.html file. You can use touch <filename> to create a new file. Create index.html in the website folder.

They usually have a CSS file as well. Create styles.css in the website folder using the same method.

List the contents of the website folder to make sure they got created.

There they are. Next is a JavaScript file. Create index.js in the website folder with the method you have been using.

You might turn this into a git repository. Create .gitignore in the website folder with the same method.

List the contents of the website folder to see your new files.

There's three files, but where's the .gitignore file? I think it's hidden. Most commands have a --help flag to show what the command can do. Display the "help" menu for the ls command. Here's an example: command <flag>

Scroll through the menu to see the flags that go with ls. The flag you are looking for is --all, or -a for short. List **all** the contents of the website folder using the correct flag.

There's the hidden file. Do you see it? It didn't display before. It also includes . and ... You used cd .. to go back a folder earlier. Change to the . directory.

You didn't go anywhere. The . takes you to the folder you are in, and .. takes you back, or up, a folder. Websites need some images. Create background.jpg in the website folder.

Next, add a header image. Create header.png in the website folder.

Finally, create footer.jpeg in the website folder.

Use the **list** command to check out the images you just added.

Looks like images show up in pink. There's also three fonts to use for the website. The first one is "roboto". Create roboto.font in the website folder.

The next one is "lato". Create lato.font in the website folder.

Lastly, create menlo.font in the website folder.

List the contents of this folder to see your new font files.

Your three font files are there. There's three icons for the website as well. Create CodeAlly.svg in the website folder.

Next, create CodeRoad.svg in the website folder.

Finally, create freeCodeCamp.svg in the website folder.

Check out the new icons you just added by listing the contents of the folder they are in.

The icons are pink as well. I think the images should go in a separate folder to clean it up a little. Make an images directory in the website folder to put them in.

List the contents of the website folder to make sure your new folder is there.

There's your new images folder. It's blue. You can copy a file with cp <file> <destination>. cp stands for "copy". Copy background.jpg to your images folder.

Better make sure it worked. Change to the images directory.

List the contents to see if background.jpg is here.

There it is. Looks like the copy worked. Change back to the website directory so you can copy the other ones.

Remind yourself of the files here by listing the contents.

You copied the background image to the images folder so you don't need the one here anymore. You can remove a file with rm <filename>. Remove background.jpg from the website folder.

List the contents to make sure it's gone.

Okay, it's gone. Next, copy header.png to the images folder.

Last, copy the "footer" image to the images folder.

All the images should be copied over. Change to the images directory so you can make sure.

Check if the images are here by listing the contents.

They all made it here. Go back to the website folder so you can delete the original files.

List the contents to remind youself of the filenames to delete.

There's two that you don't need anymore. Remove the "header" image file from the website folder since you copied to the images folder.

It should be gone. Remove the "footer" image from the website folder as well.

List the contents of the website folder to check if they are gone.

Looks like they're all deleted. There was a mistake with the extensions for the font files. You can rename them with mv like this: mv <filename> <new\_filename>. mv stands for "move", it can **rename or move** something. Rename roboto.font to roboto.woff.

Use "list" to check if it worked.

Do you see the "roboto" font? The rename worked. Next, rename the "lato" font file to lato.ttf.

Lastly, rename the "menlo" font to menlo.otf.

Use the "list" command to make sure those last two got renamed.

Take a look at the files to make sure they got renamed. Those font files could be organized into a folder as well. Make a fonts directory in the website folder to put them in.

List the contents of the website folder to make sure your new folder is there.

See it? You renamed the font files with mv, you can also move files with it. Move the "roboto" font to the new fonts folder. Here's an example: mv <file> <destination>.

You can use find to find things or view a file tree. Enter find to view the file tree of the website folder to see all the files and folders withing it.

You can see everything in this website folder and it's descendant folders. Notice that they all start with ./ to represent this folder. You can see that your font moved to the fonts folder. Next, move the "lato" font to the fonts folder.

There's one more font to move. Move the "menlo" font to the fonts folder.

Use find again to list the whole file tree and make sure those two got moved.

Yes, you can see them all in the fonts folder. Let's organize some more. Make a client directory in the website folder for the client side files.

You can make a folder in that client folder from here by adding it to the path like this: mkdir client/<new\_folder\_name>. Make a src directory in the client folder from here.

You can move files all the way across the system from here with the right path. Move index.html to the client/src folder from here.

Use find to view the file tree and make sure it moved.

Can you see the index.html file in your new src folder? Looks like it moved 😄 There's some more files that can go in the src folder. Move index.js to it from here.

Last is the CSS file. Move styles.css to the src folder.

Seems like you can do anything right from here. Take another look at the tree with find.

Things are looking more organized 😄 You can use find <folder\_name> to display the tree of a different folder. View the file tree of the client folder from the website folder.

Now you just see what's in the client folder. What else can find do? View the "help" menu of the find command to look around.

The menu isn't very pretty, but there's a -name flag in there. You can use it to search for something with find -name <filename>. Use find with the -name flag to search for index.html.

It shows you where that file is. Using the same command, find where the styles.css file is.

You can search for folders with it, as well. Using the same command and flag, find the src folder.

😄 View the file tree of the website folder to see what else you need to do.

What's next? More organizing! You should put all the assets in one spot. Change into the client folder.

Make a new directory named assets in the client folder.

Change into the new assets folder.

All the images and other assets can go here. Make an images directory in the assets folder for all the images.

Go to your new images folder.

You want the images here. Create background.jpg in this folder.

Wait. You don't need to recreate them. You can just move the other images here. Go back to the website folder from here. It's three folder back.

Now go to where the original images are. Change into the images folder.

List the contents of the images folder to see the files here.

Umm, first I think you should move them back to the website folder. Move header.png back to the website folder. The destination for the file is ..

List the contents of the images folder to see if it's gone.

It's gone. Go back to the website folder.

List what's here.

There's the file you just moved. Next, you will move it to the client/assets/images folder. First, use find with the correct flag to search for images.

There's your two image folders. Move header.png to the one with the longer path. Just use it as the destination to do so.

Use find to search for your header.png file and make sure it moved.

There it is. Right where you put it. Next, search for your footer.jpeg file so you can move that over there.

It's in the original images folder. You can **use that path** with the move command to move it. Move footer.jpeg to the client/assets/images folder while in the website folder.

View the file tree of this folder to make sure all your images are over in their new folder. Don't use any flags.

You don't need the old images folder anymore. You can use rmdir <directory\_name> to remove a folder. rmdir stands for "remove directory". Try to remove the images folder with rmdir. Make sure it's the one in the website folder.

Directory not empty? Oh yeah, there's still the background image in there. Remove the background image file in the images folder from here. Make sure it's the one in the website/images folder.

Try to remove the images folder again with rmdir. Make sure it's the one in the website folder.

I think it worked this time. List the contents to find out.

It worked, the images folder is gone. Make a new icons folder in your assets folder while in the website folder.

Move the CodeAlly.svg file to your new icons folder.

View the file tree of the website folder and make sure it moved.

Verify that the file moved to the icons folder. Next, move the "CodeRoad" file to your icons folder.

Lastly, move the "freeCodeCamp" file to your icons folder.

View the file tree and make sure the files moved.

This looks much better. The three icons are now in the icons folder. Make a fonts folder in your assets folder from here for all the font files.

Turns out you want some different fonts for the website. From here, create roboto-bold.woff in your new fonts folder. You can put the path in front of the filename of where you want it to go.

Next, create roboto-light.woff in your new fonts folder from here.

View the file tree of the client/assets/fonts folder from here to see if your new files are there.

Two more fonts to go. Create lato-bold.ttf in the new fonts folder from here.

Lastly, create lato-light.ttf in your new fonts folder from here.

View your file tree and make sure the files are there.

Things are looking more organized 😄 The new fonts are there. Now you can remove the old fonts folder and everything in it. You can't do that with rmdir since it's not empty. View the "help" menu of the rm command to see if you can find anything.

There's a -r flag that says, remove directories and their contents recursively. That will remove the folder and everything in it. Use the "remove" command with that flag to remove the fonts folder. Make sure it's the one in the website folder. Be careful not to remove the wrong folder.

List what's here to see if it's gone.

Looks like it’s gone. Surely, it went to the trash can right? No, it’s just gone. You should **be very careful when recursively removing files** like that. It will delete everything, and can destroy your operating system. There's a few more files for the boilerplate. Create package.json in the website folder.

Next, create server.js in the website folder.

Lastly, create README.md in the website folder.

List the content of this folder to make sure your new files are there.

The boilerplate is complete. Use echo to print Yay! to the terminal.

Print I finished the boilerplate! to the terminal.

Print one more thing... to the terminal

You can print to a file instead of the terminal with echo text >> filename. Use it to print I made this boilerplate to your README.md file.

Use more to view your README.md file.

Now that line is in the file. Add from the command line to your README.md file with the echo command and the same method.

Use more to view the "readme" file again.

Now the file has two lines. Add for the freeCodeCamp bash lessons to your "readme" file with the echo command like you did before.

View your "readme" file again like you did before.

😄 Change to the project folder.

You are back where you started. List what's here.

Still the same two folders. Rename the website folder to website-boilerplate.

List the contents of this folder to see the new name.

Thanks for making this. You need to make copy of it. Take a look at the "help" menu of the "copy" command.

Scroll up to find that "recursive" flag. You need to use it again to copy the whole folder. Copy the whole boilerplate into a folder named toms-website.

List the contents of the project folder to see the new copy.

Thanks. Use find to view the tree of toms-website.

Use find to view the tree of the boilerplate folder to make sure it matches.

Awesome! You are finished for now. Clear the terminal one last time.

Print "goodbye terminal" to the terminal.

Use the "exit" command to exit the terminal.

# Learn Relational Databases by Building a Mario Database

**The first thing you need to do is start the terminal.** Do that by clicking the "hamburger" menu at the top left of the screen, going to the "terminal" section, and clicking "new terminal". Once you open a new one, type echo hello PostgreSQL into the terminal and press enter.

Your virtual machine comes with PostgreSQL installed. You will use the Psql terminal application to interact with it. Log in by typing psql --username=freecodecamp dbname=postgres into the terminal and pressing enter.

Notice that the prompt changed to let you know that you are now interacting with PostgreSQL. First thing to do is see what databases are here. Type \l into the prompt to **l**ist them.

The databases you see are there by default. You can make your own like this:

CREATE DATABASE database\_name;

The capitalized words are keywords telling PostgreSQL what to do. The name of the database is the lowercase word. Note that **all commands need a semi-colon at the end.** Create a new database named first\_database.

Use the **l**ist shortcut command again to make sure your new database is there.

It worked. Your new database is there. If you don't get a message after entering a command, it means it's incomplete and you likely forgot the semi-colon. You can just add it on the next line and press enter to finish the command. Create another database named second\_database.

You should have another new database now. **L**ist the databases to make sure.

You can **c**onnect to a database by entering \c database\_name. You need to connect to add information. Connect to your second\_database.

You should see a message that you are connected. Notice that the prompt changed to second\_database=>. So the postgres=> prompt before must have meant you were connected to that database. A database is made of tables that hold your data. Enter \d to **d**isplay the tables.

Looks like there's no tables or relations yet. Similar to how you created a database, you can create a table like this:

CREATE TABLE table\_name();

Note that the parenthesis are needed for this one. It will create the table in the database you are connected to. Create a table named first\_table in second\_database.

View the tables in second\_database again with the **d**isplay command. You should see your new table there with a little meta data about it.

Create another new table in this database. Give it a name of second\_table.

There should be two tables in this database now. **D**isplay them again to make sure.

You can view more details about a table by adding the table name after the **d**isplay command like this: \d table\_name. View more details about your second\_table.

Tables need **columns** to describe the data in them, yours doesn't have any yet. Here's an example of how to add one:

ALTER TABLE table\_name ADD COLUMN column\_name DATATYPE;

Add a column to second\_table named first\_column. Give it a data type of INT. INT stands for integer. Don't forget the semi-colon. 😄

Looks like it worked. **D**isplay the details of second\_table again to see if your new column is there.

Your column is there 😄 Use ALTER TABLE and ADD COLUMN to add another column to second\_table named id that's a type of INT.

Your table should have an id column added. View the details of second\_table to make sure.

Add another column to second\_table named age. Give it a data type of INT.

Take look at the details of second\_table again.

Those are some good looking columns. You will probably need to know how to remove them. Here's an example:

ALTER TABLE table\_name DROP COLUMN column\_name;

Drop your age column.

View the details of second\_table to see if it's gone.

It's gone. Use the ALTER TABLE and DROP COLUMN keywords again to drop first\_column.

A common data type is VARCHAR. It's a short string of characters. You need to give it a maximum length when using it like this: VARCHAR(30).

Add a new column to second\_table, give it a name of name and a data type of VARCHAR(30).

Take look at the details of second\_table to see your columns.

You can see the VARCHAR type there. The 30 means the data in it can a max of 30 characters. You named that column name, it should have been username. Here's how you can rename a column:

ALTER TABLE table\_name RENAME COLUMN column\_name TO new\_name;

Rename the name column to username.

Take look at the details of second\_table again to see if it got renamed.

It worked. Rows are the actual data in the table. You can add one like this:

INSERT INTO table\_name(column\_1, column\_2) VALUES(value1, value2);

Insert a row into second\_table. Give it an id of 1, and a username of Samus. The username column expects a VARCHAR, so you need to put Samus in single quotes like this: 'Samus'.

You should have one row in your table. You can view the data in a table by querying it with the SELECT statement. Here's how it looks:

SELECT columns FROM table\_name;

Use a SELECT statement to view **all** the columns in second\_table. Use an asterisk (\*) to denote that you want to see all the columns.

There's your one row. **Insert** another row **into** second\_table. Fill in the id and username columns with the **values** 2 and 'Mario'.

You should now have two rows in the table. Use SELECT again to view **all** the columns and rows **from** second\_table.

**Insert** another row **into** second\_table. Use 3 as the id, and Luigi as the username this time.

You should now have three rows. Use SELECT again to see **all** the data you entered.

That gives me an idea 😃 You can make a database of Mario video game characters. You should start from scratch for it. Why don't you delete the record you just entered. Here's an example of how to delete a row:

DELETE FROM table\_name WHERE condition;

Remove Luigi from your table. The condition you want to use is username='Luigi'.

Luigi should be gone. Use SELECT again to see all the data and make sure he's not there.

It's gone. You can scrap all this for the new database. **delete** Mario **from** second\_table using the same command as before, except make the condition username='Mario' this time.

Only one more row should remain. **Delete** Samus **from** second\_table.

Use SELECT again to see all the rows in second\_table to make sure they're gone.

Looks like they're all gone. Remind yourself what columns you have in second\_table by looking at its **d**etails.

There's two columns. You won't need either of them for the Mario database. **Alter** the **table** second\_table and **drop** the **column** username.

Next, drop the id column.

Okay, the table has no rows or columns left. View the tables in this database to see what is here.

Still two. You won't need either of those for the new database either. Drop second\_table from your database. Here's an example:

DROP TABLE table\_name;

Next, drop first\_table from the database.

All the tables are gone now, too. View all the databases using the command to **l**ist them.

Rename first\_database to mario\_database. You can rename a database like this:

ALTER DATABASE database\_name RENAME TO new\_database\_name;

List the databases to make sure it got renamed.

**C**onnect to your newly named database so you can start adding your characters.

Now that you aren't connected to second\_database, you can drop it. Use the DROP DATABASE keywords to do that.

List the databases again to make sure it's gone.

Okay, I think you're ready to get started. I don't think you created any tables here, take a look to make sure.

Create a new table name characters, it will hold some basic information about Mario characters.

Next, you can add some columns to the table. Add a column named character\_id to your new table that is a type of SERIAL.

The SERIAL type will make your column an INT with a NOT NULL constraint, and automatically increment the integer when a new row is added. View the details of the characters table to see what SERIAL did for you.

Add a column to characters called name. Give it a data type of VARCHAR(30), and a constraint of NOT NULL. Add a constraint by putting it right after the data type.

You can make another for where they are from. Add another column named homeland. Give it a data type of VARCHAR that has a max length of 60.

Video game characters are quite colorful. Add one more column named favorite\_color. Make it a VARCHAR with a max length of 30.

You should have four columns in characters. Take a look at the details of it to see how things are going.

You are ready to start adding some rows. First is Mario. Earlier, you used this command to add a row:

INSERT INTO second\_table(id, username) VALUES(1, 'Samus');

The first parenthesis are the column names, you can put as many columns as you want. The second parenthesis are values for those columns. Add a row to your table, give it a name of Mario, a homeland of Mushroom Kingdom, and a favorite\_color of Red. Make sure to use single quotes where needed.

Mario should have a row now and his character\_id should have been automatically added. View **all** the data in your characters table with SELECT to see.

Add another row for Luigi. Give it a name of Luigi, a homeland of Mushroom Kingdom, and a favorite\_color of Green.

View all the data in your characters table with SELECT again.

Okay, it looks like it's all working. Add another row for Peach. Give her the values: Peach, Mushroom Kingdom, and Pink.

Adding rows one at a time is quite tedious. Here's an example of how you could have added the previous three rows at once:

INSERT INTO characters(name, homeland, favorite\_color)

VALUES('Mario', 'Mushroom Kingdom', 'Red'),

('Luigi', 'Mushroom Kingdom', 'Green'),

('Peach', 'Mushroom Kingdom', 'Pink');

Add two more rows. Give the first one values of: Toadstool, Mushroom Kingdom, and Red. Give the second one: Bowser, Mushroom Kingdom, and Green. Try to add them with one command.

If you don't get a message after a command, it is likely incomplete. This is because you can put a command on multiple lines. Add two more rows. Give the first one values of: Daisy, Sarasaland, and Yellow. The second: Yoshi, Dinosaur Land, and Green. Try to do it with one command.

Take a look all the data in your table with SELECT to see where you stand.

It looks good, but there's a few mistakes. You can change a value like this:

UPDATE table\_name SET column\_name=new\_value WHERE condition;

You used username='Samus' as a condition earlier. SET Daisy's favorite\_color to Orange. You can use the condition name='Daisy' to change her row.

The command you just used does exactly what it sounds like. It finds the row where name is Daisy, and sets her favorite\_color to Orange. Take a look all the data in your table again to see if she got updated.

Her favorite color was updated. Toadstool's name is wrong as well, it's actually Toad. Use UPDATE to SET his name to Toad. Use the condition favorite\_color='Red'.

Take a look all the data in your table.

Using favorite\_color='Red' was not a good idea. Mario's name changed to Toad because he likes red, and now there's two rows that are the same. Well, almost. Only the character\_id is different. You will have to use that to change it back to Mario. Use UPDATE to set the name to Mario for the row with the lowest character\_id.

Take a look all the data in your table again to see if Mario's name got changed back.

Looks like it worked. Toad's favorite color is wrong. He likes blue. Change Toad's favorite color to Blue. Use whatever condition you want, but don't change any of the other rows.

Bowser's favorite\_color is wrong. He likes Yellow. Why don't you update it without changing any of the other rows.

Bowser's homeland is wrong as well. He's from the Koopa Kingdom. Why don't you change it to that without changing any other rows.

Take a look all the data in your table again to make sure there's no more issues.

Actually, you should put that in order. Here's an example:

SELECT columns FROM table\_name ORDER BY column\_name;

View all the data again, but put it in order by character\_id.

It looks good. Next, you are going to add a **primary key**. It's a column that uniquely identifies each row in the table. Here's an example of how to set a PRIMARY KEY:

ALTER TABLE table\_name ADD PRIMARY KEY(column\_name);

The name column is pretty unique, why don't you set that as the primary key for this table.

You should set a primary key on every table and there can only be one per table. Take a look at the details of your characters table to see the primary key at the bottom.

You can see the key for your name column at the bottom. It would have been better to use character\_id for the primary key. Here's an example of how to drop a constraint:

ALTER TABLE table\_name DROP CONSTRAINT constraint\_name;

Drop the primary key on the name column. You can see the **constraint name** is characters\_pkey.

View the details of the characters table to make sure it's gone.

It's gone. Set the primary key again, but use the character\_id column this time.

View the details of the characters table to see the new primary key.

That's better. The table looks complete for now. Next, create a new table named more\_info for some extra info about the characters.

View the tables in mario\_database again with the **d**isplay command. You should have two tables now.

I wonder what that third one is. It says characters\_character\_id\_seq. I think I have a clue. View the details of the characters table.

That is what finds the next value for the character\_id column. Add a column to your new table named more\_info\_id. Make it a type of SERIAL.

Set your new column as the primary key for this table.

View the tables in mario\_database again with the display command. There should be another sequence there for the more\_info\_id because it also automatically increments.

There it is. Add another column to more\_info named birthday. Give it a data type of DATE.

Add a height column to more\_info that's a type of INT.

Add a weight column. Give it a type of NUMERIC(4, 1). That data type is for decimals. NUMERIC(4, 1) has up to four digits and one of them has to be right of the decimal.

Take a look at the details of more\_info to see all your columns.

There’s your four columns and the primary key you created at the bottom. To know what row a character is for, you need to set a **foreign key** so you can relate rows from this table to rows from your characters table. Here's an example that creates a column as a foreign key:

ALTER TABLE table\_name ADD COLUMN column\_name DATATYPE REFERENCES referenced\_table\_name(referenced\_column\_name);

That's quite the command. In the more\_info table, create a character\_id column. Make it an INT and a foreign key that references the character\_id column from the characters table. Good luck.

To set a row in more\_info for Mario, you just need to set the character\_id (foreign key) value to whatever it is in the characters table. Take a look at the details of more\_info to see your foreign key.

There's your foreign key at the bottom. These tables have a "one-to-one" relationship. **One** row in the characters table will be related to exactly **one** row in more\_info and vice versa. Enforce that by adding the UNIQUE constraint to your foreign key. Here's an example:

ALTER TABLE table\_name ADD UNIQUE(column\_name);

Add the UNIQUE constraint to the column you just added.

The column should also be NOT NULL since you don't want to have a row that is for nobody. Here's an example:

ALTER TABLE table\_name ALTER COLUMN column\_name SET NOT NULL;

Add the NOT NULL constraint to your foreign key column.

Take a look at the details of your more\_info table to see all the keys and constraints you added.

The structure is set, now you can add some rows. First, you need to know what character\_id you need for the foreign key column. You have viewed all columns in a table with \*. You can pick columns by putting in the column name instead of \*. Use SELECT to view the character\_id column **from** the characters table.

That list of numbers doesn't really help. Use SELECT again to display both the character\_id and name columns from the characters table. You can separate the column names with a comma to view both.

That's better. You can see Mario's id there. Here's some more info for him:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1981-07-09 | 155 | 64.5 |

Add a row to more\_info with the above data for Mario using the INSERT INTO and VALUES keywords. Be sure to set his character\_id when adding him. Also, DATE values need a string with the format: 'YYYY-MM-DD'.

View all the data in more\_info to make sure it's looking good.

Next, you are going to add some info for Luigi. Use SELECT again to view the character\_id and name columns **from** the characters table to find his id.

You can see Luigi's id there. Here's his info:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1983-07-14 | 175 | 48.8 |

Add a row in more\_info for Luigi using the above info. Be sure to add his character\_id as well.

View all the data in more\_info to see more info for Luigi.

Peach is next. View the character\_id and name columns from the characters table again so you can find her id.

Here's the additional info for Peach:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1985-10-18 | 173 | 52.2 |

Add a row for Peach using the above info. Be sure to add her character\_id as well.

Toad is next. Instead of viewing all the rows to find his id, you can just view his row with a WHERE condition. You used several earlier to delete and update rows. You can use it to view rows as well. Here's an example:

SELECT columns FROM table\_name WHERE condition;

A condition you used before was username='Samus'. Find Toad's id by viewing the character\_id and name columns from characters for only his row.

Here's what Toad's info looks like:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1950-01-10 | 66 | 35.6 |

Add the above info for Toad. Be sure to add his character\_id.

View all the data in more\_info to see the rows you added.

Bowser is next. Find his ID by viewing the character\_id and name columns for only his row.

Here's what Bowser's info looks like:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1990-10-29 | 258 | 300 |

Add the above info for Bowser. Don't forget to add his character\_id.

Daisy is next. Find her ID by viewing the character\_id and name columns for only her row.

The info for Daisy looks like this:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1989-07-31 | NULL | NULL |

Add the above info for Daisy to more\_info. Be sure to add her character\_id as well. You can use NULL or simply not include the null columns when inserting.

View all the data in more\_info to see the rows you added.

Null values show up as blank. Yoshi is last. Find his ID by viewing the character\_id and name columns for only his row.

The info for Yoshi looks like this:

| **birthday** | **height** | **weight** |
| --- | --- | --- |
| 1990-04-13 | 162 | 59.1 |

Add the above info for Yoshi to more\_info. Be sure to include his character\_id.

There should be a lot of data in more\_info now. Take a look at **all** the rows and columns in it.

It looks good. There is something you can do to help out though. What units do the height and weight columns use? It's centimeters and kilograms, but nobody will know. Rename the height column to height\_in\_cm.

Rename the weight column to weight\_in\_kg.

Take a quick look at all the data in more\_info to see the new column names.

Next, you will make a sounds table that holds filenames of sounds the characters make. You created your other tables similar to this:

CREATE TABLE table\_name();

Inside those parenthesis you can put columns for a table so you don't need to add them with a separate command, like this:

CREATE TABLE table\_name(column\_name DATATYPE CONSTRAINTS);

Create a new table named sounds. Give it a column named sound\_id of type SERIAL and a contraint of PRIMARY KEY.

View the tables in mario\_database to make sure it worked.

There's your sounds table. Add a column to it named filename. Make it a VARCHAR that has a max length of 40 and with constraints of NOT NULL and UNIQUE. You can put those contraints at the end of the query to add them all.

You want to use character\_id as a foreign key again. This will be a "one-to-many" relationship because **one** character will have **many** sounds, but no sound will have more than one character. Here's the example again:

ALTER TABLE table\_name ADD COLUMN column\_name DATATYPE CONSTRAINT REFERENCES referenced\_table\_name(referenced\_column\_name);

Add a column to sounds named character\_id. Give it the properties INT, NOT NULL, and set it as a foreign key that references character\_id from characters.

Take a look at the details of the sounds table to see all the columns.

Next you will add some rows. But first, view all the data in characters so you can find the correct id's again. **Order** them **by** character\_id like you did earlier.

The first file is named its-a-me.wav. Insert it into the sounds table with Mario's id as the character\_id.

Add another row with a filename of yippee.wav. Use Mario's character\_id again for the foreign key value.

Add another row to sounds for Luigi named ha-ha.wav. Use his character\_id this time. Take a look at the data in characters to find his id if you need to.

Add another row with a filename of oh-yeah.wav. This one is for Luigi as well so use his character\_id again.

Add two more rows for Peach sounds. The filenames are yay.wav and woo-hoo.wav. Don't forget her character\_id. Try to do it with one command.

Add two more rows. The filenames are mm-hmm.wav and yahoo.wav. The first one is for Peach again, the second is for Mario, so use the correct foreign key values. Try to do it with one command.

View all the data in the sounds. You should be able to see the "one-to-many" relationship better. One character has many sounds.

See the "one-to-many" relationship? Create another new table called actions. Give it a column named action\_id that's a type of SERIAL, and make it the PRIMARY KEY. Try to create the table and add the column with one command.

Add a column named action to your new table. Give it a type of VARCHAR that is a max length of 20 and has UNIQUE and NOT NULL constraints.

The actions table won't have any foreign keys. It's going to have a "many-to-many" relationship with the characters table. This is because **many** of the characters can perform **many** actions. You will see why you don't need a foreign key later. Insert a row into the actions table. Give it an action of run.

Insert another row into the actions table. Give it an action of jump.

Add another action row with an action of duck.

View all the data in actions to make sure there's no mistakes.

It look good. "Many-to-many" relationships usually use a **junction** table to link two tables together, forming two "one-to-many" relationships. Your characters and actions table will be linked using a junction table. Create a new table called character\_actions. It will describe what actions each character can perform.

Your junction table will use the primary keys from the characters and actions tables as foreign keys to create the relationship. Add a column named character\_id to your junction table. Give it the type of INT and constraint of NOT NULL.

The foreign keys you set before were added when you created the column. You can set an existing column as a foreign key like this:

ALTER TABLE table\_name ADD FOREIGN KEY(column\_name) REFERENCES referenced\_table(referenced\_column);

Set the character\_id column you just added as a foreign key that references the character\_id from the characters table.

View the details of the character\_actions table to see the foreign key you added.

Add another column to character\_actions named action\_id. Give it a type of INT and constraint of NOT NULL.

This will be a foreign key as well. Set the action\_id column you just added as a foreign key that references the action\_id column from the actions table.

View the details of the character\_actions table to see your keys.

Every table should have a primary key. Your previous tables had a single column as a primary key. This one will be different. You can create a primary key from two columns, known as a **composite** primary key. Here's an example:

ALTER TABLE table\_name ADD PRIMARY KEY(column1, column2);

Use character\_id and action\_id to create a composite primary key for this table.

This table will have multiple rows with the same character\_id, and multiple rows the same action\_id. So neither of them are unique. But you will never have the same character\_id and action\_id in a single row. So the two columns together can be used to uniquely identify each row. View the details of the character\_actions table to see your composite key.

Insert three rows into character\_actions for all the actions Yoshi can perform. He can perform all of them in the actions table. View the data in the characters and actions table to find the correct id's for the information.

View all the data in character\_actions to see your rows.

Add three more rows into character\_actions for all of Daisy's actions. She can perform all of the actions, as well.

Bowser can perform all the actions. Add three rows to the table for him.

Next is Toad. Add three more rows for his actions.

You guessed it. Peach can perform all the actions as well, so add three more rows for her.

Add three more rows for Luigi's actions.

Last is Mario, add three rows for his actions.

That was a lot of work. View all the data in character\_actions to see the rows you ended up with.

Well done. The database is complete for now. Take a look around to see what you ended up with. First, display all the tables you created.

There's five tables there. Nice job. Next, take a look at all the data in the characters table.

Those are some lovely characters. View all the data in the more\_info table.

You can see the character\_id there so you just need to find the matching id in the characters table to find out who it's for. Or... You added that as a foreign key, that means you can get all the data from both tables with a JOIN command:

SELECT columns FROM table\_1 FULL JOIN table\_2 ON table\_1.primary\_key\_column = table\_2.foreign\_key\_column;

Enter a join command to see **all** the info from both tables. The two tables are characters and more\_info. The columns are the character\_id column from both tables since those are the linked keys.

Now you can see all the info from both tables. If you recall, that's a "one-to-one" relationship. So there's one row in each table that matches a row from the other. Use another JOIN command to view the characters and sounds tables together. They both use the character\_id column for their keys as well.

This shows the "one-to-many" relationship. You can see that some of the characters have more than one row because they have **many** sounds. How can you see all the info from the characters, actions, and character\_actions tables? Here's an example that joins three tables:

SELECT columns FROM junction\_table

FULL JOIN table\_1 ON junction\_table.foreign\_key\_column = table\_1.primary\_key\_column

FULL JOIN table\_2 ON junction\_table.foreign\_key\_column = table\_2.primary\_key\_column;

Congratulations on making it this far. This is the last step. View all the data from characters, actions, and character\_actions by joining all three tables. When you see the data, be sure to check the "many-to\_many" relationship. Many characters will have many actions.

# Learn Bash Scripting by Building Five Programs

**The first thing you need to do is start the terminal.** Do that by clicking the "hamburger" menu at the top left of the screen, going to the "terminal" section, and clicking "new terminal". Once you open a new one, type echo hello bash into the terminal and press enter.

You can run commands in the terminal or put them in a file to be run as a script. You will be making five small programs to learn some scripting. The first one will be a "questionnaire". Use the touch command to create questionnaire.sh in the project folder.

To start, open the file in the main editor by clicking the filename in the left side panel.  
Then, add the text echo hello questionnaire at the top of the file.

Your script has one command. Run it with sh questionnaire.sh to see what happens. sh stands for shell.

Using sh to run your script uses the shell interpreter. Run your script again with bash questionnaire.sh to use the bash interpreter. bash stands for bourne-again shell.

The output was the same. There are many interpreters which may not give the output you expect. Find out where the bash interpreter is located by entering which bash in the terminal.

That's the absolute path to the bash interpreter. You can tell your program to use it by placing a shebang at the very top of the file like this: #!<path\_to\_interpreter>. Add a shebang at the very top of your file, the one you want looks like this: #!/bin/bash.

Now, instead of using sh or bash to run your script. You can run it by executing the file and it will default to bash. Run your script by executing it with ./questionnaire.sh

You should have got a permission denied message because you don't have permissions to execute the script. List what's in the project folder in long list format with ls -l to see the file permissions.

Next to your file is -rw-r--r--. All but the first character (-) describe permissions different users have with the file. r means read, w means write, x means execute. I don't see an x anywhere, so nobody can execute it. Enter chmod +x questionnaire.sh in the terminal to give everyone executable permissions.

List what's in the folder again with ls -l to see the new permissions.

The x was added by each type of user to denote that anyone can execute the file. Run your file again by executing it with ./questionnaire.sh.

Now it works. In your script, you can add any commands that you would be able to enter in the terminal. Test this by adding the ls -l command below your other command.

Run the script by executing it again.

Your script printed the result of the two commands as if you entered them in the terminal. Delete everything but the shebang from your file so you can start making the questionnaire.

Bash has variables, functions, and other things you might be familiar with. You can create a variable with VARIABLE\_NAME=VALUE. There cannot be any spaces around the equal (=) sign. If a variable has any spaces in it, place double quotes around it. Create a variable named QUESTION1 and set it's value to "What's your name?".

To use a variable, place $ in front of it like this: $VARIABLE\_NAME. Shell scripts run from top to bottom, so you can only use variable below where it's created. Use echo to print your variable.

Run the file like you did before to see if it worked.

The question was printed. Next, you want to be able to accept input from a user. You can do that with read like this: read VARIABLE\_NAME. This will get user input and store it into a new variable. After you print the question, use read to get input and store it in a variable named NAME.

At the bottom of your script, use echo to print Hello <name>. to the terminal.

Run the file again. Type your name and press enter after it asks for it.

Right below your first variable, create another one named QUESTION2. Set the value to, Where are you from?. Make sure to put it in double quotes.

After your read command, use your new variable to print the next question.

Below where the second question is printed, use read to get input from the user into a variable named LOCATION.

Change the existing response to Hello <name> from <location>..

Run the script and enter values when it is waiting for input.

It's looking good. I want a title to appear when the program first starts. Use echo to print ~~ Questionnaire ~~ before anything else is printed.

Run the script and enter values until it is done again so you can see what the title looks like.

It would be nice if there was some empty lines around the title. You've probably used the --help flag before, see if you can use it with echo to try and find a way to add empty lines.

That didn't work as I hoped. Another way to find information about a command is with man. It stands for manual and you can use it like this: man <command>. See if there's a manual for echo.

At the top of the menu, the -e option looks promising. And the \n below it says new line. You should take a look at those. In your script, change the title to echo -e \n~~ Questionnaire ~~\n to see if that prints the empty lines.

Run it to see if it worked. You can press ctrl+c to close the program after it starts if you don't want to enter values.

It didn't print the empty lines. echo will only print empty lines if the value is enclosed in quotes. Place double quotes around the title that gets printed to see if it works.

Run your script again to see if that fixed it.

Now it's working 😄 Create a QUESTION3 variable next to the other two, set it's value to "What's your favorite coding website?"

Use echo to print the third question after you read the LOCATION.

After the question you just printed, add code to read input into a variable named WEBSITE.

Change the echo command of the response to print this sentence instead: Hello <name> from <location>. I learned that your favorite coding website is <website>!.

Run the script and enter values when the program is waiting. Let's see the final output.

One last thing. Change that final response to print an empty line at the beginning of the sentence.

Run it one last time and enter values when it asks to see if you like how it looks.

It looks good. I think you are done with that script for now. The next program will be countdown timer. Use the touch command to create a new file named countdown.sh in your project folder.

Give your file executable permissions so you can run it like the other one. It's the chmod command with the +x flag.

You want to use the bash interpreter again. Add a shebang at the top of your new file to denote that.

Comments in bash look like this: # <comment>. Add a comment below the shebang that says Program that counts down to zero from a given argument so people know what it does. Note that the shebang is a special case and is not treated like a comment.

Programs can take arguments. You can access them a few different ways with $. Add echo $\* in your script to print all arguments passed to it.

Execute your script with ./countdown.sh.

Nothing was printed. Run your script again, but this time add three arguments to the command; arg1, arg2, and arg3. Place them after the command with a space before each one.

$\* printed all the arguments passed to your script. To access any one of them, use $<number>. arg2 could have been accessed with $2. Change your script to echo the first argument instead of all the arguments.

Run your file with ./countdown.sh arg1 arg2 arg3 again.

Now it just prints the first argument. Your program will accept an argument to count down from. You will test it with an if statement to make sure it's a positive integer. I wonder what that syntax would look like. Type help in the terminal to see if you can find anything.

This is a list of built-in commands. You should look over it, some of them may look familiar. I see echo in there. Another one is if. See if you can find out more about it by checking its man page.

I guess there isn't a man page for it. At the top of the help screen, I noticed you can use help <command> to find out more. Yet another way to find out about a command 😥 See if you can find out more about if with that method.

The syntax is at the top, not all of it is required. Here's another example:

if [[ CONDITION ]]

then

STATEMENTS

fi

Remove the echo $1 in your script and add an if condition that checks if [[ $1 == arg1 ]]. In its then area, use echo to print true to the screen. There must be spaces on the inside of the brackets ([[ ... ]]) and around the operator (==).

Notice that the end of the syntax is fi (if backwards). It should print true if you pass arg1 to your script now. Run the script with arg1 as the only argument.

The if condition worked, it printed true. Run it again with anything except arg1 as the first argument.

Nothing was printed. One of the optional parts of if was an else area. You can use it like this:

if [[ CONDITION ]]

then

STATEMENTS

else

STATEMENTS

fi

Add an else to your existing if condition. Use echo to print false if the condition fails.

Run the script again and use anything except arg1 as the only argument.

Now it printed false. Your program is expecting an integer to count down from as its argument. You can compare integers inside the brackets ([[ ... ]]) of your if with -eq (equal), -ne (not equal), -lt (less than), -le (less than or equal), -gt (greater than), -ge (greater than or equal). Change your if condition to check if your first argument is less than 5.

Run the script again and use 4 as a first argument to make sure it's working.

It printed true since your argument was less than 5. Run it again with 5 as the argument.

As expected, that printed false. Take a look at that help menu again. I want to see if we can find out more about how these expressions work.

Near the top left, it says [[ expression ]]. Those look like the double brackets you are using. See if you can get more info about that with the help command like you did with help if.

It might not be a bad idea to read that. Looks like you can use some, probably familiar, things like !, &&, and || to compare multiple expressions. There's also == and != operators for an individual expression. It says something about the test built-in command. See if you can bring up the help menu for that.

That's what I was looking for. At the top are some file operators. There's some string and other operators as well. You should take a look at them. Near the bottom, are the arithmetic operators you used with your if condition. Change the condition in your script to check if the first argument is less than or equal to 5.

Run the script and use 5 as a first argument again.

Now it prints true. Remember I said any command can run in the terminal or a script. Try running an expression right in the terminal by entering [[ 4 -le 5 ]] in it.

Nothing happened? Each command has an exit status that can be accessed with $?. View the exit status of the **last command** with echo $?.

The exit status of 0 means it was true, 4 is indeed less or equal to 5. Try it again with [[ 4 -ge 5 ]].

Use echo to view the exit status of the command you just entered.

It printed 1 this time for false. You can separate commands on a single line with ;. Enter your last two commands on one line like this: [[ 4 -ge 5 ]]; echo $?. It will run the expression, then print the exit status of it since it was the last command.

It's still false. Using the same syntax of [[ ... ]]; echo$?, check if 10 is not equal to 5 and print the exit status of the expression on one line.

You can think of an exit status of 0 as true. But it means that the command had zero errors. All commands have an exit status. Using the same syntax, enter bad\_command; and check its exit status on a single line.

command not found, with an exit status of 127. Anything but 0 means there was an error with the command. bad\_command didn't exist. Try it again with ls.

The command executed as expected and there were zero errors. So it gave you an exit status of 0. Try it again with ls -y.

The -y flag doesn't work with ls so it gave you an exit status other than 0, meaning that the command was unsuccessful. View the help menu of the test command again, I want to see what else is in that list.

You tried a few of the arithmetic operators, those work for integers. Try one of the file operators. The first one on the list checks if a file exists. Type [[ -a countdown.sh ]]; echo $? in the terminal to see if your file exists.

The file must exist. It's checking the folder the command is entered from. Try it again with bad\_file.txt.

bad\_file.txt doesn't exist. I think you're getting the hang of this. Using the same syntax, check if you have permissions to execute your countdown.sh file. You may want to look at that menu again.

You played around with a number of the expressions. View the help [[ expression ]] menu again that you looked at before to see a few more options. You can view the menu with just help [[.

As I mentioned before, you can test multiple expressions with && and ||. Enter [[ -x countdown.sh && 5 -le 4 ]]; echo $? in the terminal to test the file is executable by you **and** five is less than or equal to four.

Both conditions weren't true, so the exit status was 1 for false. Try testing the same two conditions with the or operator.

One of the conditions was true so it printed 0. I think that's enough of a detour. Back in your script, change the if condition to check if the first argument is **greater than zero** so you can be sure it's something you can count down from.

The condition you added checks if a positive integer was passed as an argument to the script and executes the then area. Change the existing echo command to print Include a positive integer as the first argument. if a positive integer is not used.

Run your script and use 1 as a first argument to make sure the condition is working.

Run it again and use anything but a positive integer as the only argument.

Looks like your if condition is working. Next, you want to loop over the argument and count down to zero from it. Check the help menu to see if there's any commands for this.

There's two for loops in there, you want the second one. Here's an example:

for (( i = 10; i > 0; i-- ))

do

echo $i

done

The above creates a variable (i = 10), then prints it, subtracts one, and repeats until i is not greater than 0. So it prints 10 through 1. In the then area of your condition, replace the echo with a for loop that prints from the argument ($1) to 1.

Run your script and use 10 as the first argument.

It works 😄 But I want it to pause for one second between each number. Check the help menu again to see if there's any commands that might help.

I'm not seeing the command I was hoping to. These are the built-in commands, where are the rest? Type ls / to look around.

The / listed what's in the root of the file system. I see a bin folder, bin stands for binary. View what's in it with ls /bin.

These are some non built-in commands. There's quite a few that should look familiar. One is bash, that's the one you used for the shebang in your scripts. I see one called sleep. View the manual of it.

At the top, it says you can pause execution for a number of seconds. Try it out by entering sleep 3 in the terminal.

That should work. In your for loop, use sleep to make the script pause for 1 second after each number is printed.

Run your script and use 3 as the first argument.

Awesome. Except it should print 0 instead of stopping at 1. Change the condition in your for loop so that it checks for i >= 0.

Run your script with 3 as the argument again.

Excellent. I want it to display a title like the other script. Make it so that it prints ~~ Countdown Timer ~~ before anything else. Include a new line before and after it like you did for the other title.

Run your script and use 1 as the first argument again to see the title.

This is fun. You can create a multiline comment like this:

: '

comment here

more comment here

'

Comment out your for loop with a multiline comment. I want to try and do this with a while loop.

View the help menu for the while command to see if you can find anything.

It shows the syntax. First, below your comment, create a variable named I that is set to the value of your first argument. It will start there, then on each iteration of the while loop you can subtract 1 from it until it reaches 0.

The menu showed that you can make a while loop like this:

while [[ CONDITION ]]

do

STATEMENTS

done

Add a while loop below the I variable you made. The condition should be $I -ge 0 and you should echo the I variable in the do statements.

I never changes here, so you would have an infinite loop. You can subtract one from I with double parenthesis (((...))) and the -- operator. In your while loop, add (( I-- )) after you echo $I to subtract one from I on each pass.

The last thing to do is to add the sleep again. In your while loop, add the code to make it sleep for 1 second. Add the code after the (( I-- )).

Run the script and use 5 as the first argument.

I think the countdown timer is finished. Feel free to try it with some other arguments. The next one is a bingo number generator. Use touch to create bingo.sh in the same folder as the others.

Give your file executable permissions like you did for the other two.

Add a shebang at the top of your new script. It should use bash again like other two.

Add a comment below the shebang that says, Bingo Number Generator.

Before I forget, use a single echo command to print a title for this program. It should say ~~ Bingo Number Generator ~~ with an empty line before and after it.

In your script, create a NUMBER variable that equals 5.

Below your new variable, use echo to print it to the screen.

Run the script by executing it.

The numbers in bingo go up to 75, each number has a letter from the word bingo associated with it. You will need to randomly generate a number between 1 and 75. Bash may have something that can help you here. A shell comes with environment variables. View them by entering printenv in the terminal.

These are all environment variables, they are predefined and loaded with each shell. Most of them aren’t very relevant, but it’s nice to know they’re there. One of them is LANG. Use echo to print it in the terminal.

View all variables in the shell with declare -p. -p stands for print

This list includes all the environment variables, and any others that may have been created in the current shell. There's one named RANDOM. Use echo to print it in the terminal.

Back in your script, use the RANDOM variable to set NUMBER to a random number instead of 5.

Run the script a few times in a row to make sure it's working.

The RANDOM variable will generate a random number between 0 and 32767. You can use the modulus operator to make it in the range you want. In your script, change the NUMBER variable to $RANDOM%75.

Run the script again.

Bash sees everything as a string so it just printed the %75 part literally. In the terminal, create an I variable equal to 0 (zero), so you can play with it and figure out how to do some calculations.

In the terminal, use echo to print your new variable.

I noticed that you used double parenthesis in the while loop of your countdown timer to subtract one from I. Type (( I++ )) in the terminal to see if anything happens.

There was no output. Use echo to print I in the terminal again.

The double parenthesis performed the calculation, changing the value of I from 0 to 1. Enter help let in the terminal to see the operators you can use with the double parenthesis.

You used several of these now, including in the for loop from the countdown timer. Enter (( I += 10 )) in the terminal to increment I by 10. Note that you don't need to prepend variables with $ inside these parenthesis.

Use echo to print your I variable again.

It should have printed 11 for the value of I. Using the double parenthesis like you have been is good for changing variable values or making comparisons. It makes the calculation in place and provides no output. If you want to make a calculation and do something with the result, add a $ in front like this: $(( ... )). Type $(( I + 4 )) in the terminal to see what happens.

It should say, bash: 15: command not found. It replaced the command with the result of the calculation. Effectively, trying to run 15 as a command. Enter the same command, but put echo in front of it. The command was $(( I + 4 ))

Again, it replaced the calculation with the result. So it was basically the same as if you entered echo 15. Use echo to print I to the screen again.

It should still have printed 11 for I. See the hints if it didn't. These double parenthesis with a $ are how you can assign a variable to some calculation. In the terminal, create a J variable, and use the $(( ... )) syntax to set its value to I - 6.

Use echo to print J.

J should equal 5. For some more practice, use echo to print the value J \* 5 + 25.

It should have printed 50. Print J with echo again.

So, as a reminder, (( ... )) will perform a calculation or operation and output nothing. $(( ... )) will replace the calculation with the result of it. You made a few variables in this shell, view them with declare -p.

declare can be used to create variables, but you are just going to use it to view them for now. If you scroll up a little, you should find your I and J variables in there. View J with declare -p J.

I saw RANDOM in that list, too. View it with declare -p <variable> like you did for J.

Okay, I think I finally know how to get the random number for the Bingo Number Generator. Use echo and RANDOM % 75 to print a random number in the terminal.

One tiny problem, that calculation will give a number between 0 and 74. Enter the same command in the terminal, but add 1 to the calculation to get a random number between 1 and 75.

Back in your bingo.sh script, change the NUMBER variable so that it starts as a random number between 1 and 75 using the syntax you have been practicing.

Run your script a few times in a row to make sure it's working.

Next, create a TEXT variable and set the value to "The next number is, ". When the script is finished, the output will be something like The next number is B:15.

The letter that goes with the random number depends on what the number is. If it's 15 or less, it will be a B. I saw some comparisons in the help let menu, take a look at it again.

You used the double square brackets with your if statement in the last program, but you can use the double parenthesis with these operators as well. In your script, create an if statement that uses double parenthesis for the condition. Check if the number variable is less than or equal to 15. If it is, use your two variables to print The next number is, B:<number>.

if statements can have an "else if" area like this:

if (( CONDITION ))

then

STATEMENTS

elif [[ CONDITION ]]

then

STATEMENTS

fi

Using the double square brackets this time, add an elif condition that checks if the number variable is less than or equal to 30. If it is, use your two variables again to print The next number is, I:<number>

You can add as many elif sections to an if statement as you want. Add another elif, below the last, one that uses the double parenthesis to check if the number variable is less than 46. If it is, use your two variables to print The next number is, N:<number>

Run your script if you want to see the output. It should print one of the sentences if the random number is less than 46. It may take a couple tries. Add another elif, below the last one, that uses double square brackets to check if the number variable is less than 61. If it is, use your two variables to print The next number is, G:<number>

One more case to handle. Add an else at the bottom of the if that uses your two variables to print The next number is, O:<number>.

Run your script a few times and make sure it's working.

I think the generator is done 😄 The next project is a fortune teller. Use the touch command to create fortune.sh in the same folder as the other scripts.

Give your file executable permissions.

Add a shebang at the top of your new file that uses bash again.

Add comment Program to tell a persons fortune

Add a title for this one like the others. This one should say ~~ Fortune Teller ~~. Don't forget the empty line before and after it.

Run the file once to make sure it's working.

This program will have an array of responses. One will be printed randomly after a user inputs a question. Practice first 😄 In the terminal, create an array like this: ARR=("a" "b" "c")

Each variable in the array is like any other variable, just combined into a single variable. In the terminal, print the second item in the array with echo ${ARR[1]}. Note that the first item would be index zero.

If you recall, you were able to print all the arguments to your countdown.sh script with echo $\*. echo $@ would have worked as well. Similarly, you can use the \* or @ to print your whole array. In the terminal, use echo to print all the items in your array.

The variable must be in that declare list. View your array variable using the declare command and the -p flag.

The -a next to it stands for array. In your script, create an array named RESPONSES. Give it these six values: Yes, No, Maybe, Outlook good, Don't count on it, and Ask again later.

In your script, use echo to print the last item in the array.

Run it to see the output.

You will randomly print one of the values. In your script, create a variable named N. Set it equal to a random number between 0 and 5, the first and last index of the array.

Change your echo command to print the item in the array whose index is the random number you generated.

You will create a function to generate an answer. Check the help menu to see if you can find anything.

See any that might help? There's one that says function. See if you can find out more about it.

It looks like you can create a function like this:

FUNCTION\_NAME() {

STATEMENTS

}

Add an empty function named GET\_FORTUNE to your script. Make sure the repsonse you are printing is the last thing in the script.

In your function, use echo to print Ask a yes or no question:

Call your function by putting the name of it below where you create it. No $ needed. Make sure the reponse you are printing is at the bottom of the file.

Run your script to make sure it's working.

In your function after you print the sentence, use read to get user input into a variable named QUESTION.

Run the script again to test it out. Enter a question when it asks.

I want to make sure the input is a question. You are going to add a loop that asks for input until the input ends with a question mark. View the help menu to see if you can find an appropriate loop.

View more about that until command. That might be the one to use here.

The until loop is very similar to the while loop you used. It will execute the loop until a condition is met. Here's an example:

until [[ CONDITION ]]

do

STATEMENTS

done

Add an until loop below your function. Use the double brackets to check if QUESTION is equal to test?. Move the GET\_FORTUNE function call to the statements area of the loop. It should run the function until you input test? as the question.

Run the script and enter something other than test?. Then enter test? after it asks for a question the second time.

View that help [[ expression ]] menu again. You need to find out how to test if the input ends with a question mark (?).

Let's play with these again. You can test if two strings are the same with ==. In the terminal, use the [[ ... ]]; echo $? syntax you used before to test if hello is equal to hello.

Exit status of 0, it was true. Using the same syntax, test if hello is equal to world.

False. An import operator in that menu is =~. It allows for pattern matching. Using the same syntax but with this operator, check if hello contains the pattern el.

True. The condition was checking for el within the word hello. Using the same syntax, check if hello world contains the pattern lo wor. You will need to put them both in quotes so it recognizes the spaces.

Your patterns have been checking for literal matches, el and lo wor. You can use regular expression characters as well, but you can't put the pattern in quotes when you do. Using the same syntax, check if hello world starts with an h by using ^h as the pattern.

Do it again, but use ^h.+d$ as the pattern to see if the string starts with an h, has at least one character after it, and ends with a d.

In the terminal, create a variable named VAR that equals hello world.

Use echo to print the variable you just created.

Using the [[ ... ]]; echo $? syntax, check if your variable is equal to hello world.

Using the same syntax, check if your variable ends with ? by using the pattern \?$.

It doesn't end with ?. Just to make sure I don't have the pattern wrong, check if test? ends with ?.

I think that will work. Back in your script, change the until condition to see if your variable ends with ?.

Run the script and input something that doesn't end with ? the first time, then something that does the second.

I know that it asks the same thing if the input isn't what you want. You should let users know that it needs to end with ?. Add an if condition in your **function** that checks if [[ ! $1 ]]. Put the existing echo statement in the then area and make sure the existing read is below the whole if condition.

You can pass arguments to functions like you did with your script. This condition will check if one isn't passed and print the sentence. Add an else to your if. Use echo to print Try again. Make sure it ends with a question mark: if the condition fails.

Now, your function will print one thing if you pass it any argument, and something else if not. In the until loop, add again as an argument to where you call the function.

Now, each time the function is called in the until loop, it will pass again as an argument and print the Try again... sentence. Before your until loop, call the function without an argument so the first time it runs, it prints the initial sentence.

Run the script and enter something without a question mark when it asks the first time. Use a question mark the second time.

Awesome. One last thing. Add an empty line in front of where you print the response.

Run the script one more time to see if you like the output.

Excellent. One last program to make. Use touch to create a new file named five.sh in the same folder as the others.

Give your file executable permissions.

Add a shebang to the new script that uses bash like the others.

Add a comment below the shebang that says, Program to run my other four programs

This program will run all the programs you made so far consecutively. Add the command to run the questionnaire.sh file.

Run the file to see if it works. Enter input when it asks.

Add commands to run the rest of your scripts in the file. They should be in this order: questionnaire, countdown, bingo, and fortune. Don't forget that your countdown.sh file needs an argument, so put a 3 next to it.

Okay, use clear to empty out what's in the terminal before the big moment.

Run the script and enter input when it asks.

Cool. I think all the scripts are done. View the help menu again I want to explore one more thing.

View more about that type command.

It says you can view the type of a command with type <command>. Just for fun, lets take a look at the type of a few different commands. View the type of echo.

View the type of the read command.

View the type of if

View the type of then

Those were all from the help menu and described as a shell builtin or shell keyword. View the type of bash

That's the location of the bash command. View the type of psql.

It's showing the location of the commands. View the type of your ./five.sh file.

Last step, close the terminal with the exit command. Thanks and happy coding!

# Learn SQL by Building a Student Database: Part 1

**The first thing you need to do is start the terminal.** Do that by clicking the "hamburger" menu at the top left of the screen, going to the "terminal" section, and clicking "new terminal". Once you open a new one, type echo hello SQL into the terminal and press enter.

You are started with two .csv files with info about your computer science students. You should take a look at them. The top row in each file has titles, and the rest are values for those titles. You will be adding all that info to a PostgreSQL database. Log into the psql interactive terminal with psql --username=freecodecamp --dbname=postgres to get started.

View the existing databases with the \l shortcut command to see what's here.

All the info from the CSV files will go into a single database. Create a new database named students.

View the databases again to make sure it got created.

There it is. Connect to your new database so you can start adding tables.

The CSV files have a bunch of students with info about them, and some courses and majors. You will have four tables. One for the students and their info, one for each major, another for each course, and a final one for showing what courses are included in each major. First, create the students table.

The second table will be for each unique major that appears in the data. Create a table named majors.

The third table is for each unique course in the data. Create another table named courses.

The final table will be a junction table for the majors and courses. Create it with the name majors\_courses.

Use the **d**isplay shortcut command to view your tables to make sure your satisfied with them.

Onto the columns. The students.csv file has four fields, you will make a column for each of those as well as an ID column. Add a column to your students table named student\_id. Give it a type of SERIAL so it automatically increments and make it a PRIMARY KEY

The first column in students.csv is first\_name. Add a column to the students table with that name. Make it a type of VARCHAR(50) and give it the NOT NULL constraint.

The next column in the data is last\_name. Add it to the students table. Give it the same data type and max-length as first\_name and make sure it has the NOT NULL constraint.

The next column is for the major. Since you will have each major in another table this column will be a foreign key that references it. Create a column in the students table named major\_id, give it a data type of INT for now. You will come back and set the foreign key later.

Create the last column, gpa. The data in the CSV shows that they are decimals with a length of 2 and 1 number is to the right of the decimal. So give it a data type of NUMERIC(2,1).

Use the shortcut command to **d**isplay the details of the students table to make sure you like it.

The foreign key is still missing. Let's fill in the majors table next. Add a major\_id column to it. Make it a type of SERIAL and the PRIMARY KEY for this table.

This table will only have one other column for the name of the major. Add a column to it named major. Make it a VARCHAR with a max-length of 50 and give it the NOT NULL constraint.

View the details of the majors table to make sure you like it.

This table looks good. Now, set the major\_id column from the students table as a foreign key that references the major\_id column from the majors table. Here's an example of how to do that: ALTER TABLE <table\_name> ADD FOREIGN KEY(<column\_name>) REFERENCES <referenced\_table\_name>(<referenced\_column\_name>);

View the details of the students table again to make sure the key is there.

Next, is the courses table. Add a course\_id column to it. Give it a type of SERIAL and make it the primary key.

Add a course column to the courses table that's a type of VARCHAR. The course names are a little longer, so give them a max-length of 100. Also, make sure it can't accept null values.

View the details of the courses table to make sure it looks good.

One more table to go. The majors\_courses junction table will have two columns, each referencing the primary key from two related table. First, add a major\_id column to it. Just give it a type of INT for now.

Set the major\_id column you just created as a foreign key that references the major\_id column from the majors table.

Next, add a course\_id column to the same table. Just give it a type of INT again for now.

Set your new course\_id column as a foreign key that references the other course\_id column.

View the details of the table you just worked on to make sure the structure is finished.

There's one thing missing. This table doesn't have a primary key. The data from courses.csv will go in this table. A single major will be in it multiple times, and same with a course. So neither of them can be a primary key. But there will never be a row with the same two values as another row. So the two columns together, are unique. You can create a composite primary key that uses more than one column as a unique pair like this: ALTER TABLE <table\_name> ADD PRIMARY KEY(<column\_name>, <column\_name>); Add a composite primary key to the table using the two columns.

View the details of the table again.

Okay, now it's finished. View all the tables you ended up with.

Next, you can start adding some info. Since the students table needs a major\_id, you can add a major first. View the details of the majors table to see what info it expects.

It only needs the name of a major. The ID will be added automatically. Add the first major from the courses.csv file into the majors table. It's a VARCHAR, so make sure to put the value in single quotes.

Use SELECT to view all the data in the majors table to make sure it got inserted correctly.

Next, insert the first course from courses.csv into the courses table.

View all the data in the courses table to make sure it got added.

Next, you can add a row into the junction table. View the details of it to see what it expects.

It wants a major\_id and course\_id. Add a row to majors\_courses for the first entry in courses.csv.

View all the data in the table you just added to.

Looks like the row got added. View the details of the students table to remind yourself what it expects so you can add the first student to the database.

The output shows what the table needs. Insert the first person from students.csv into the students table.

Looks like it worked. View all the data in the students table to make sure.

Okay, you added a row into each table. It might be wise to review the data and the database structure. Adding the rest of the info one at a time would be tedious. You are going to make a script to do it for you. I recommend "splitting" the terminal for this part. You can do that by clicking the "hamburger" menu at the top left of the window, going to the "Terminal" menu, and clicking "Split Terminal". Once you've done that, use the touch command to create a file named insert\_data.sh in your project folder.

You should have two terminals open. One connected to PostgreSQL, and one for entering terminal commands. In the one for terminal commands, use the chmod command with the +x flag to give you new script executable permissions.

Open your new file and add a "shebang" that uses bash at the top. It looks like this: #!/bin/bash.

Below that, add a single line comment with the text, Script to insert data from courses.csv and students.csv into students database.

First, you should add all the info from the courses.csv file since you need the major\_id for inserting the student info. cat is a terminal command for printing the contents of a file. Here's an example: cat <filename>. Below the comment you added, use it to print courses.csv.

Run your script to see if the file contents get printed.

It worked. Instead of printing the content, you can pipe that output into a while loop so you can go through the rows one at a time. It looks like this:

cat courses.csv | while read MAJOR COURSE

do

<STATEMENTS>

done

Each new line will be read into the variables, MAJOR and COURSE. Add the above to your cat command. In the STATEMENTS area, use echo to print the MAJOR variable.

Run the script to see if it worked.

It's looping, but the MAJOR variable is only being set to the first word. There's a default IFS variable in bash. IFS stands for "Internal Field Separator". View it with declare -p IFS.

This variable is used to determine word boundaries. It defaults to spaces, tabs, and new lines. This is why the MAJOR variable was set to only the first word on each line from the data. Between the while and read commands, set the IFS to a comma like this: IFS=","

Now, it should use the comma in the data to separate words instead of spaces. Run the script again to see if it's working.

Looks like that worked. It prints the whole major, including the space. Print the COURSE variable on the same line as where you print MAJOR to make sure it's all working.

Run the script again to check.

Okay, your loop is working. You can use the MAJOR and COURSE variables to access the major or course when you need to insert data or query the database. Delete the echo line so you can figure out what to do next.

It helps to plan out what you want to happen. For each loop, you will want to add the major to the database if it isn't in there yet. Same for the course. Then add a row to the majors\_courses table. Add these single line comments in your loop in this order: get major\_id, if not found, insert major, get new major\_id, get course\_id, if not found, insert course, get new course\_id, insert into majors\_courses.

You used the psql command to log in and interact with the database. You can use it to just run a single command and exit. Above your loop, add a PSQL variable that looks like this: PSQL="psql -X --username=freecodecamp --dbname=students --no-align --tuples-only -c". This will allow you to query your database from your script. The important parts are the username, dbname, and the -c flag that is for running a single command and exiting. The rest of the flags are for formatting.

Now, you can query your database using the PSQL variable like this: $($PSQL "<query\_here>"). Below the get major\_id comment in your loop, create a MAJOR\_ID variable. Set it equal to the result of a query that gets the major\_id of the current MAJOR in the loop. Make sure to put your MAJOR variable in single quotes.

Below the variable you just created, use echo to print it so you can see it's value when you run the script.

Run the script to see what happens.

So it went through each major from the CSV file and tried to find major\_id for each one from the database. Looks like it only found the one you manually inserted earlier. The rest were empty. Below your first if not found comment, add an if condition that checks if the MAJOR\_ID variable is empty. You can do that with this test: [[ -z $MAJOR\_ID ]]. Place the next two comments in the statements area of the if.

The loop will go into this if whenever a major isn't found. Here, you will want to insert the major and then get the new id. You will need the ID for inserting data into the majors\_courses table later. Below your insert major comment, create an INSERT\_MAJOR\_RESULT variable. Set it's value to a query that inserts the current major into the database. Don't forget to use single quotes around the value.

Below the variable you just created, use echo to print it.

Instead of running through all the data in the CSV file, you should make some test data. In the terminal, use the copy (cp) command to copy the courses.csv into a new file named courses\_test.csv.

In your new file, remove all the data except for the first five lines. Make sure there's a single empty line at the bottom.

Back in the insert\_data.sh script, change your cat command to loop through the test file instead of the full one.

Run the script. It will go through the test data and insert a major into the database each time it doesn't find one already there and print the MAJOR\_ID and INSERT\_MAJOR\_RESULT variables.

Looks like found an ID that was already in the database twice and inserted three new items into the database. You don't need to print the ID anymore so delete the echo $MAJOR\_ID line.

In the psql prompt, use SELECT to view all the data from the majors table to see what the script added.

I forgot you inserted Database Administration earlier. The script ran and inserted major from the top line of the file. Then it added the other two that weren't already in there. You can use TRUNCATE to delete all data from a table. In the psql prompt, try to delete all the data in the majors table by entering TRUNCATE majors;

It says you "cannot truncate a table referenced in a foreign key constraint." The students and majors\_courses tables use the major\_id from majors as a foreign key. So if you want to delete the data from majors, you need to delete the data from those two tables at the same time. Use TRUNCATE to delete the data from those three tables. Separate the tables with commas.

View all the data in the majors table to make sure it's empty.

Looks like it worked. View all the data in the majors\_courses table to see if that one is empty.

It is, check the students table.

Last, check the courses table.

There should still be one entry in there. Use TRUNCATE to delete all the data from the courses table. You will need to truncate any tables that use a column from it as a foreign key at the same time.

View all the data in the courses table again.

Now the database is completely empty. Run the script again to see what gets inserted when the database is empty.

It inserted four that time. In the psql prompt, view all the data in the majors table.

You won't want to add the first line from the CSV file to the database since those are just titles. In your script, add an if condition at the top of your loop that checks if $MAJOR != major. Put all the existing code and comments in your loop in it's statements area so it only does any of it if it's not the first line.

In the psql prompt, use TRUNCATE to delete all the data in the majors table.

View all the data in majors table to make sure it's empty.

Run the script to make sure it's not adding the first line anymore.

It only showed three inserts, that's a good sign. View all the data in majors table to make sure it's three you want.

There's three unique majors in your test data. Those were the three added to the database, so it looks like it's working. Delete the line where you print INSERT\_MAJOR\_RESULT.

You want a nicer message when something get's inserted so it's more informative. Below your INSERT\_MAJOR\_RESULT variable, add an if statement that checks if the variable is equal to INSERT 0 1, which was what it was printing. Use echo to print Inserted into majors, $MAJOR in the statements area of the if.

In the psql prompt, truncate the majors table again so you can run the script and see the output.

Check to make sure the table is empty. Then, run the script.

It's starting to come together. Below your get new major\_id comment, set the MAJOR\_ID variable to a query that gets the new major\_id from the database.

So the script will insert the majors correctly. Next are the courses. It will be the same steps as for the majors. Below your get course\_id comment, add a COURSE\_ID variable that gets the course\_id from the database. Remember that your COURSE variable will have the current course in the loop.

It's the same as the majors, so below the second if not found comment, add an if statement that checks if the query was empty so you can insert the course if needed. Place the existing insert course and get new course\_id comments in the statements area of the if.

Below the insert course comment, create an INSERT\_COURSE\_RESULT variable that inserts the course into the database.

The variable should be INSERT 0 1 again if something gets inserted. Below the variable you just created, add an if condition that checks if it is and print Inserted into courses, $COURSE using echo in it's statements area.

In the psql prompt, truncate the data from the majors table so you can run the script again.

Run the script to see if the courses get inserted into the database.

It looks like it worked. The test data has three unique courses, and three got added to the database. View the data in the courses table to make sure they are correct.

Excellent. Instead of manually deleting the data each time you want to run the script, add the command to do it for you. Near the top of the file below your PSQL variable, use echo to query the database. In the query, truncate your four tables in this order: students, majors, courses, majors\_courses.

Run the script to see if it works.

Awesome. That makes it easier. Below your get new course\_id comment, set the COURSE\_ID to the newly inserted course\_id.

One more thing to add for this file. Below the insert into majors\_courses courses comment, create a INSERT\_MAJORS\_COURSES\_RESULT variable. Use it and the MAJOR\_ID and COURSE\_ID variables you created to insert a row into the majors\_courses table. Make sure the query has the major\_id column first. Also, you won't need any quotes around the values for the ID's.

Below the variable you just created, add an if condition that checks if it's equal to INSERT 0 1 like the others. In it's statements area, use echo to print Inserted into majors\_courses, $MAJOR : $COURSE.

Run the script. Your tables should get truncated and then it should go through the loop and add all the data from the courses\_test.csv into the three tables of the database.

Looks like it works. You better look around to make sure. View the data in the majors table.

Cool, check the courses table.

Lastly, view the data in the majors\_courses table. There should be four rows.

Alright, that part of the script is done. Next, you need to add everything from the students.csv file. Make some test data again. In the terminal, use the copy command to copy students.csv into a file named students\_test.csv.

In the students\_test.csv file, remove everything but the first five lines like you did for the other test file. Make sure there's an empty line at the bottom again.

You want to loop through all this info like you did for the other CSV file. The process is the same. Below your existing loop, use cat to print your new test file. Pipe the results into a while loop, setting the IFS to a comma again, and then use read to create FIRST, LAST, MAJOR and GPA variables from the data. In the loop, use echo to print the FIRST variable.

Run the script to see if it prints the FIRST (first\_name) variable correctly. It will take a second since it has to go through the first loop.

It works 😅 It printed the first item in each row of the CSV file. It's printing the first line again, you will have to take care of that. First, delete the echo line.

Add an if condition to the loop that checks if the FIRST variable is not equal to first\_name so it doesn't do anything for the first line of the file. Don't put anything in the statements area for now.

All the columns in the CSV file can be inserted directly into the database except for the major. You will need to get the major\_id again for that. There's some null values in there as well, so you will need to use null if the major\_id isn't found. Add four single line comments in your loop; get major\_id, if not found, set to null, and insert student in that order.

Below the new get major\_id comment, set the MAJOR\_ID variable to a query that gets the major\_id for the current students major.

Below that, use echo to print the variable so you can see if it's working.

Run the script to see what happens.

Looking at the test data, it found the ID for all of it except the null value. Below the newest if not found comment, add an if that checks if the variable is empty. Put the set to null comment in its statements area.

When you go to insert the student data, you want to use the MAJOR\_ID if it's found, or null if not. Below the set to null comment, set the MAJOR\_ID variable to null so you can use it to insert the data.

Move the echo $MAJOR\_ID line to below the if statement so you can run the script and see the value of the variable if the major\_id is or isn't found.

Run the script.

Okay, that should work for inserting the student. Delete the echo $MAJOR\_ID line.

One last thing to add. In the psql prompt, view the details of the students table so you can see what columns to add.

You will need to set the four columns when adding the student info. All of them except student\_id. Below the insert student comment, create an INSERT\_STUDENT\_RESULT variable that adds the student to the database. Add the columns in the order they appear in the data, and make sure to only put the two VARCHAR columns in single quotes.

Below the variable you just created, add an if statement that checks if it's equal to INSERT 0 1 like the others. If it is, use echo to print Inserted into students, <first\_name> <last\_name>.

Run the script to see if the students are getting added.

I think it's working. View all the data in the students table to make sure it matches the CSV file.

Excellent. It added all the students from the test data. Time to try it with the original files. Change the cat courses\_test.csv line to use the original file again.

Next, change the cat students\_test.csv line to use the original file as well.

Time for the moment of truth. Run the script and see if it works.

That was cool. View all the data in the students table to see what you ended up with.

31 rows. That's how many are in the CSV file. Perfect. Next, check the majors table.

7 rows. There must be 7 unique majors in the CSV file. View what's in the courses table.

Looks like there's 17 unique courses in the CSV file. Last, view the data in majors\_courses. This should have the same number of rows at the CSV file.

28 rows, same as the CSV file. I think all the data got added correctly. You don't need your test files anymore. In the terminal, use the list command to check what files are in your project folder.Use the remove command (rm) to delete the students\_test.csv file.Use the same command to delete the courses\_test.csv file.List the contents of the folder again to make sure they're gone.The database is finished for now. The last thing you are going to do is make a "dump" of it. The pg\_dump command can do that for you. Use the --help flag with the command to see what it can do.This is the last step. There's quite a few options there. Enter pg\_dump --clean --create --inserts --username=freecodecamp students > students.sql in the terminal to dump the database into a students.sql file. It will save all the commands needed to rebuild it. Take a quick look at the file when you are done.

# Learn SQL by Building a Student Database: Part 2

**The first thing you need to do is start the terminal.** Do that by clicking the "hamburger" menu at the top left of the screen, going to the "terminal" section, and clicking "new terminal". Once you open a new one, type echo hello SQL into the terminal and press enter.

In Part 1 of this tutorial, you created a students database and then a script to insert information about your computer science students into it. Log into the psql interactive terminal with psql --username=freecodecamp --dbname=postgres to see if it's here.

List the databases.

Your database isn't here. You can use the .sql file you created at the end of Part 1 to rebuild it. I recommend "splitting" the terminal. You can do that by clicking the "hamburger" menu at the top left of the window, going to the "Terminal" menu, and clicking "Split Terminal". Once you've done that, enter psql -U postgres < students.sql in it to rebuild the database.

A lot of stuff happened in the terminal. That looks promising. In the psql prompt, view the databases again.

There's your students database. Connect to it.

Now that you're connected. Display the tables and relations that are here to see if it's all correct.

That all looks right. View the details of the students table to make sure the stucture is right.

Looks good. Make sure all the data is in the table, as well.

The data is all there. You should take a look at the details of the other tables and the data in them to make sure they look good. When you are done, use touch in the bash terminal to create student\_info.sh. You are going to make a script to print info about your students.

Give your new file executable permissions.

Add a shebang that uses bash at the top of your new script.

Below the shebang, add a comment that says Info about my computer science students from students database.

In the new script, use echo to print ~~ My Computer Science Students ~~. Use the -e flag with it to put a new line at the beginning and end of the text.

Run the script to make sure it's working.

You will want to query the database again to get info about the students to display. Add the same PSQL variable you use in your insert\_data.sh script. It looked like this: PSQL="psql -X --username=freecodecamp --dbname=students --no-align --tuples-only -c"

Below the PSQL variable you just added, use echo to print First name, last name, and GPA of students with a 4.0 GPA:. Use the -e flag to put a new line at the beginning of the sentence.

You will want to print what that sentence is asking for. You should know how to make that query, but lets practice a little first. SQL stands for "Structured Query Language". It's the language you have been using to manage your relational databases. In the psql prompt, view all the data in the students table like you have done many times.

You should look at the column titles that were returned. The \* gets all columns in a table with your query. You can return specific columns by putting the column name in the query instead of \*. In the psql prompt, view just the first\_name column from the students table.

Just the first\_name column was returned that time. You can specify as many columns you want returned by separating them with commas. View the first\_name, last\_name and gpa columns from the students table.

You can return only rows you want by adding WHERE <condition> to your query. A condition can consist of a column, an operator, and a value. Use one of these to view the same columns as before but only rows WHERE gpa < 2.5.

The < only return rows where the gpa column was less than 2.5. Some other operators are: <, >, <=, >=. View the same columns, but only rows for students with a gpa greater than or equal to 3.8.

That only returned students with a GPA of 3.8 or better. There's equal (=) and not equal (!=) operators as well. View the same columns for students that don't have a 4.0 gpa.

The right query will get you only the data you are looking for. Back in your student\_info.sh file, add an echo command to the bottom that prints what the sentence above it asks for. Place double quotes around it like this: echo "$($PSQL "<query\_here>")". This will make it so the output isn't all on one line.

Run the script to see your students with the highest GPA's.

Add another echo statement at the bottom of the script. Make it print All course names whose first letter is before 'D' in the alphabet:. Put a new line in front of it like the first sentence.

Practice first. In the psql prompt, view all the data in the majors table.

The operators you used with numbers in the last section can be used on text as well. Use the = to view all majors named Game Design. Don't forget that You need single quotes around text values.

Next, view all the rows not equal to Game Design.

Use the greater than operator to see majors that come after it alphabetically.

Game Design was not included in the results because it is not > 'Game Design'. Try it with the greater than or equal to operator.

It included Game Design in the results that time. So if you want to see results that start with a G or after, you could use major >= 'G'. View the majors that come before G.

In your script, add an echo at the bottom to print the suggested info like you did before. Make sure to use double quotes where needed.

Run the script to see what course names come before the letter D.

Looks like there is five of them. Add another sentence like the others that says: First name, last name, and GPA of students whose last name begins with an 'R' or after and have a GPA greater than 3.8 or less than 2.0:

To find that, start by using the psql prompt to view all the data in the students table.

It returned 31 rows. Use the same command, but only return the rows for students whose last name comes before M in the alphabet.

That returned 18 rows. You can use multiple conditions after WHERE with AND or OR, among others. Just add the keyword and another condition. In the psql prompt, use the same command as before, but add an OR to also return rows of students with a 3.9 GPA.

It showed rows where one of the conditions was true, there was one more than last time. Enter the previous command, but use AND to view only students that meet both conditions.

Now it only shows rows where both conditions are true, one person. Enter the previous command, but add a third condition of OR gpa < 2.3.

This showed all students whose GPA is less than 2.3 because the final OR condition was true for them. It didn't matter what their last name started with. You can group conditions together with parenthesis like this: WHERE <condition\_1> AND (<condition\_2> OR <condition\_2>). This would only return rows where <condition\_1> is true and one of the others is true. View students whose last name is before M that have a GPA of 3.9 or less than 2.3.

Two students meet those conditions. Back in the student info file, add an echo command at the bottom to print the suggested rows.

Run the script to see the results.

Moving along. Add another echo command, like the others, with a sentence that says: Last name of students whose last name contains a case insensitive 'sa' or have an 'r' as the second to last letter:

Start by viewing everything from the courses table in the psql prompt to see how you might be able to find this out.

There's a few that contain the word Algorithms. You can use LIKE to find patterns in text like this: WHERE <column> LIKE '<pattern>'. An underscore (\_) in a pattern will return rows that have any character in that spot. View the rows in this table with a course name that matches the pattern '\_lgorithms'.

That pattern matched only rows that had exactly one character, followed by lgorithms. Another pattern character is %. It means anything can be there. To find names that start with W, you could use W%. View the courses that end in lgorithms.

It found two that time. Try viewing courses that start with Web.

Combine the two pattern matching characters to show courses that have a second letter of e.

Nice job! Try viewing the courses with a space in their names.

There they are. You can use NOT LIKE to find things that don't match a pattern. View courses that don't contain a space.

Five courses without a space. Try finding the ones that contain an A.

6 rows. This showed all the courses with a capital A. ILIKE will ignore the case of the letters when matching. Use it to see the courses with an A or a.

It found 11 rows that time. You can put NOT in front of ILIKE as well. Use it to see the courses that don't contain an A or a.

You combine these like any other conditions. View the courses that don't have a capital or lowercase A and have a space.

In your student info script, add an echo statement at the bottom like the other to print the results of the suggested query.

Run the script to see the results.

Looks like five students meet those conditions. Add another echo command at the bottom, like the others. Make this one say: First name, last name, and GPA of students who have not selected a major and either their first name begins with 'D' or they have a GPA greater than 3.0:

Start by looking at all the data in the students table.

All the fields that are empty or blank are null. You can access them using IS NULL as a condition like this: WHERE <column> IS NULL. View the students who don't have a GPA.

Inversely, you can use IS NOT NULL to see rows that aren't null. View all the info on students that do have a GPA.

View all the info on students who haven't chosen a major.

View the students who don't have a major, but don't include students without a GPA.

One more. View the students who don't have a major and gpa.

In your script, add an echo command at the bottom to print the results the sentence is looking for.

Run the script to see the students that meet those conditions.

There's three of them. Add another sentence, like the others that says Course name of the first five courses, in reverse alphabetical order, that have an 'e' as the second letter or end with an 's':

You can specify the order you want your results to be in by adding ORDER BY <column\_name> at the end of a query. In the psql prompt, view all the info in the students table in order by the GPA's.

That put the lowest GPA's at the top. When using ORDER BY, it will be in ascending (ASC) order by default. Add DESC (descending) at the end of the last query to put the highest ones at the top.

Now, the highest GPA's are at the top. You can add more columns to the order by separating them with a comma like this: ORDER BY <column\_1>, <column\_2>. Any matching values in the first ordered column will then be ordered by the next. View all the student info with the highest GPA's at the top, and in alphabetical order by first\_name if the GPA's match.

Many times, you only want to return a certain number of rows. You can add LIMIT <number> at the end of the query to only get the amount you want. View the students in the same order as the last command, but only return the first 10 rows.

The order of the keywords in your query matters. You cannot put LIMIT before ORDER BY, or either of them before WHERE. View the same number of students, in the same order, but don't get the ones who don't have a GPA.

In your script, add the echo command to print the rows the sentence is asking for.

Run the script to see the courses.

😎 Add another echo command at the bottom of the script like the others. Make this one say, Average GPA of all students rounded to two decimal places:

There's a number of mathematic functions to use with numerical columns. One of them is MIN, you can use it when selecting a column like this: SELECT MIN(<column>) FROM <table>. It will find the lowest value in the column. In the psql prompt, view the lowest value in the gpa column of the students table.

Another one is MAX, use it to see the largest gpa of the same table.

In the same fashion, use a SUM function find out what all the values of the major\_id column in the students table add up to.

AVG will give you the average of all the values in a column. Use it to see the average of the same column.

You can round decimals up or down to the nearest whole number with CEIL and FLOOR, respectively. Use CEIL to round the average major\_id up to the nearest whole number. Here's an example: CEIL(<number\_to\_round>).

Or, you can round a number to the nearest whole number with ROUND. Use it to round the average of the major\_id column to the nearest whole number.

You can round to a specific number of decimal places by adding a comma and number to ROUND, like this: ROUND(<number\_to\_round>, <decimals\_places>). Round the average of the major\_id to five decimal places.

You should be able to find what your script is asking for now. Add the command to print it.

Run the script to see the average GPA of all your students.

They're doing pretty good. Add another command to print Major ID, total number of students in a column named 'number\_of\_students', and average GPA rounded to two decimal places in a column name 'average\_gpa', for each major ID in the students table having a student count greater than 1:

Another function is COUNT. You can use it like this: COUNT(<column>). It will tell you how many entries are in a table for the column. Try it out in the psql prompt by using COUNT(\*) to see how many majors there are.

Using the same method, check how many students you have.

Using \* like that told you how many total rows are in the table. View the count of the major\_id column in the students table to see how many of your students have picked a major.

Using major\_id didn't count the null values in that column. 23 students have a major. DISTINCT is a function that will show you only unique values. You can use it like this: DISTINCT(<column>). View the unique major\_id values in the students table.

There's six unique major\_id values in the students table. You can get the same results with GROUP BY. Here's an example of how to use it: SELECT <column> FROM <table> GROUP BY <column>. Use this method to view the unique major\_id values in the students table again.

The output was the same as DISTINCT, but with GROUP BY you can add any of the aggregate functions (MIN, MAX, COUNT, etc) to it to find more information. For instance, if you wanted to see how many students were in each major you could use SELECT COUNT(\*) FROM students GROUP BY major\_id. View the major\_id column **and** number of students in each major\_id.

When using GROUP BY, any columns in the SELECT area must be included in the GROUP BY area. Other columns must be used with any of the aggregate functions (MAX, AVG, COUNT, etc). View the unique major\_id values with GROUP BY again, but see what the lowest GPA is in each of them.

Nice job. Enter the same query, but add a column that shows you the highest GPA in each major as well.

Another option with GROUP BY is HAVING. You can add it at the end like this: SELECT <column> FROM <table> GROUP BY <column> HAVING <condition>. The condition must be an aggregate function with a test. An example to might be to use HAVING COUNT(\*) > 0 to only show what whatever column is grouped that have at least one row. Use HAVING to only show rows from the last query that have a maximum GPA of 4.0.

Two of your majors have at least one student with a 4.0 GPA. Looking at the results, the column is named min. You can rename a column with AS like this: SELECT <column> AS <new\_column\_name> Enter the same command, but rename the min column to min\_gpa.

Now the column has a better name. Enter the same command, but rename the max column to max\_gpa as well.

That's more descriptive. View the major\_id and number of students in each major\_id in a column named number\_of\_students.

Use HAVING with the last query to only show the rows with less than eight students in the major.

Well done. Back in your script, add the command the print the suggested results.

Run the script to see the output.

Add an echo command to your script like the others that prints List of majors, in alphabetical order, that either no student is taking or has a student whose first name contains a case insensitive 'ma':

The majors and students table are linked with the major\_id foreign key. If you want to see the name of a major that a student is taking, you need to JOIN the two tables into one. Here's an example of how to do that:  
SELECT \* FROM <table\_1> FULL JOIN <table\_2> ON <table\_1>.<foreign\_key\_column> = <table\_2>.<foreign\_key\_column>;

In the psql prompt, join the two tables together with the above method.

It's showing all the columns from both tables, the two major\_id columns are the same in each row for the ones that have it. You can see that there are some students without a major, and some majors without any students. The FULL JOIN you used will include **all** rows from both tables, whether or not they have a row using that foreign key in the other. From there, you could use any of the previous methods to narrow down, group, order, etc. Use a LEFT JOIN to join the same two tables in the same way.

There's a few less rows than the last query. In the LEFT JOIN you used, the students table was the left table since it was on the left side of the JOIN. majors was the right table. A LEFT JOIN gets all rows from the left table, but only rows from the right table that are linked to from the left one. Looking at the data, you can see that every student was returned, but the majors without any students were not. Join the same two tables with a RIGHT JOIN this time.

The right join showed all the rows from the right table (majors), but only rows from the left table (students) if they have a major. There's one more type you should know about. Join the two tables with an INNER JOIN.

The INNER JOIN only returned students if they have a major and majors that have a student. In other words, it only returned rows if they have a value in the foreign key column (major\_id) of the opposite table. You should know a little about the four main types of joins now. Try using a LEFT JOIN to show **all the majors** but only students that have a major.

Excellent. All the majors are there. Next, use the appropriate join to show only students that are enrolled in a major, and only majors that have a student enrolled in it.

👍 Try using a right join to show all students but only majors if a student is enrolled in it.

That showed all the students since it was the right table of the RIGHT JOIN. Use the appropriate join with the same two table to show all rows in both tables whether they have a value in the foreign key column or not.

Lets do some more experiments with joins. Say you wanted to find a list of majors that students are taking. Use the most efficient JOIN to join the two tables you need. Only join the tables for now, don't use any other conditions.

Good. To get the list, you don't need all the columns, though. Enter the same command, but just get the column you need.

You also don't want any duplicates. Use DISTINCT to only return the unique ones to see the list of majors who have students.

There's your list of majors that students are taking 😄 Next, say you wanted a list of majors that students aren't taking. Use the most efficient JOIN to join the two tables you need. Only join the tables for now, don't use any other conditions.

That got you all the majors, you can see the ones that don't have any students. Add a WHERE condition to only see the majors without students, use student\_id in it's condition.

Now you only have the rows you need. Only get the columns you need with it to see the list of majors without students.

You're doing great. Next, use the most efficient 'JOIN' to join the tables you would need if you were asked to get the first name, last name, major, and GPA of students who are taking Data Science or have a gpa of 3.8 or greater. Only join the tables for now, don't use any other conditions.

Enter the same command, but use WHERE to only get the students that meet the requirements. As a reminder, the goal was to find students who are taking Data Science or have a gpa of 3.8 or greater.

Now, you have narrowed it down the rows you are looking for. Enter the same command, but only get the columns you need. There was four of them, the students first name, last name, their major, and GPA. Get them in that order.

From there, you could put them in a specific order if you wanted or limit the results to a certain number among other things. Lastly, use the most efficient 'JOIN' to join the tables you would need if you were asked to get the first name and major for students whose first\_name, or the major, contains ri. Only join the tables for now, don't use any other conditions.

Add a WHERE to the previous query so you only get the rows you need. The rows you wanted were the ones with a first name or major containing ri.

Finally, you only wanted to display the first\_name and major columns. Enter the previous query, but only get the columns you need.

In your script, add the command to print what the sentence is asking for.

Run the script to see the majors described.

😄 Almost done. In your script, add a command to print this sentence like the others: List of unique courses, in reverse alphabetical order, that no student or 'Obie Hilpert' is taking:

Lets go over a few more things before you figure out how to see the courses a student is taking. Start by doing a FULL JOIN on your students and majors tables.

If you look at the column names, it shows two major\_id columns. One from the students table and one from the majors table. If you were to try and query it using major\_id, you would get an error. You would need to specify what table you want the column from like this: <table>.<column>. Enter the same join but only get the major\_id column from the students table.

Earlier, you used AS to rename columns. You can use it to rename tables, or give them aliases, as well. Here's an example: SELECT \* FROM <table> AS <new\_name>;. Enter the same query you just entered, but rename the majors table to m. Anywhere the majors table is referenced, you will need to use m instead of majors.

This doesn't affect the output. It can just make some queries easier to read. Enter the same query, but rename the students table to s as well.

There's a shortcut keyword, USING to join tables if the foreign key column has the same name in both tables. Here's an example: SELECT \* FROM <table\_1> FULL JOIN <table\_2> USING(<column>);. Use this method to see **all** the columns in the students and majors table. Don't use any aliases.

Note that the two major\_id columns were turned into one with USING. In order to find out what courses a student is taking, you will need to join all the tables together. You can add a third table to a join like this: SELECT \* FROM <table\_1> FULL JOIN <table\_2> USING(<column>) FULL JOIN <table\_3> USING(<column>). This example will join the first two tables into one, turning it into the left table for the second join. Use this method to join the two tables from the previous query with the majors\_courses table.

You may need to adjust the terminal size to align the output. What you're seeing is every unique combination of rows in the database. Students with a major are listed multiple times, one for each course included in the major. The majors without any students are there along with the courses for them. The students without a major are included, they have no courses and are only listed once. You can join as many tables together as you want. Join the last table to the previous command to get the names of the courses with all this info.

Same amount of rows, but you get the course names now. In your script, add the command to print the suggested info.

Run the script to see courses described.

Last one. Add a command that prints List of courses, in alphabetical order, with only one student enrolled:.

Go for it.

This is the last step, you have done really well. Run the script one last time. 👋

pg\_dump -cC --inserts -U freecodecamp worldcup > worldcup.sql