codecademy

Off Platform Project: Designing A Database From Scratch

In learning about database schemas, we have always given you some guiding information. Whether you were working off of pictures of the schema itself, or a description of the information that we needed to store in a database, you were always building your database schema with our help. For this project, we're stepping all the way back and letting you design the database from scratch.

Welcome! By now you should be familiar with the concept of a database schema and the components that go into a schema. To briefly review, those components are:

- Tables with columns of a particular data type
- Primary Keys to uniquely identify items in your tables
- Foreign Keys to define relationships between two tables
    - There are many types of relationships you can define with keys. Namely, one-to-one, one-to-many, and many-to-many.

For this project, there will be a "research" component where you will have to learn what data is even relevant for your database. We will discuss this more in the sections below. For now, know that the main purposes of this project are:

- To design a database schema on a topic of your choosing.
- To implement that schema on your own computer using Postbird.

Let's get started!

Part 1 — Research

For this project, you are going to create a database based on a topic of your choosing. To make things even trickier, this project works best when you create a database scheme based on a topic that you don't already have knowledge about. How do you even begin to start a project like this? This is where the "research" portion of this project comes into play.

Ask a friend to describe a system that they work with that you are unfamiliar with. As you are interviewing them, think about how the system they are describing can be translated into a database.

- Are there specific entities or objects in this system? Those objects will probably be translated into tables.
- Are there relationships between these entities? What types of relationships are those? One-to-one? One-to-many? Many-to-many? Those relationships will be translated into foreign keys.
- Is there a way to uniquely identify entities? If so, that should be represented by primary keys.

As you work through this research phase, your end goal should be a Schema Diagram, also known as an Entity-Relationship Diagram. There are many online tools, like dbdiagram.io and sqldbm.com that can help you create these diagrams, but there's nothing wrong with sketching this diagram out by hand!

As you interview your friend and learn about the database that you will be building, have them give you input on your diagram. You don't need to explain the technical details of the diagram to them, but you should confirm that your understanding of the relationship between entities matches what they're describing to you.

We strongly suggest interviewing a friend to learn about a subject matter that is new to you. However, if you are having trouble coming up with a topic, these are some topics that might work that you could use your existing knowledge of:

- Schools: What are the relationships between teachers, classes, sections, students, schools, years, etc. How would you represent these relationships in a Database Schema?
- Movies: How would all of the pieces of making a movie be represented in a database? Think about entities like directors, actors, movies, studios, movie theaters, etc.
- Music: This one is similar to movies. What entities exist in the music industry that could be represented in a database? Consider singers, bands, agents, producers, studios, etc.

We did this exercise ourselves! You should find an image of our Entity-Relationship Diagram in the files that you downloaded. If you want to read through our process of creating our own Entity-Relationship Diagram, you can find our notes at the bottom of this document.

Part 2 — Turn Your Diagram Into A Database Schema

Now that you've learned about the database that you'll be creating, it's time to implement your database in Postgres! We used Postbird to connect to our local PostgreSQL server, but you can use whatever client you are most comfortable with.

There are a number of steps that go into this process. We'll try to highlight some of the most important ones below.

- It's probably best to create a new database for this project. That way, you won't have any tables from other projects in your database. If you are using Postbird, you can create a new database in the "Select Database" dropdown menu in the top left corner of the UI.
- Create the tables that you've drawn up in your Entity-Relationship Diagram. As you create these tables, there are a few things to watch out for.
    - Make sure you've given each column an appropriate data type.
    - Define primary keys so each row in your tables is uniquely identifiable.
    - Define foreign keys to create relationships between tables. The ordering here might be a little tricky. If you're creating a table with a foreign key referencing a column from a different table, make sure that the other table has already been created!

Once again, you can go to the bottom of this document to see some of our thoughts on this process when we did this exercise.

Part 3 — Add Data To Your Tables

Now that you've created your tables, one of the best ways to verify they're working as expected is to add some test data. Add enough rows to your tables to test all of the relationships you have created. For example, if you think you have created a one-to-many relationship using foreign keys, make sure that you can actually create multiple rows that all reference a single row from another table.

Part 4 — Edit Your Schema As Necessary

As you entered data into your tables, you might have uncovered some issues that you didn't anticipate. That's ok! It's hard to get a schema perfect on the first try. Maybe a column's data type needs to be changed. Or maybe you realized that a one-to-many relationship should actually be a many-to-many relationship. Whatever the problem is, this is your opportunity to iterate.

If you're not familiar with how to edit a table's structure after it has already been created, you can take a look at the documentation on ALTER TABLE. Specifically, some of the commands that might be useful are `ADD COLUMN`, `DROP COLUMN`, `SET DATA TYPE`, and `ADD table_constraint`.

Reflection

Nice work! We hope that by going through this project you're gaining more confidence in your ability to design a database on your own. This is a task that you might encounter as a real database engineer! The ability to collaborate with non-technical teammates to understand their needs and translate those needs into a PostgreSQL database is such a valuable skill to develop. Now that you've done this exercise once, start thinking about other systems that you use on a day to day basis and how those systems might be represented as a database!

Our Database

In this section, we'll walk you through our thought process as we tried this exercise. We'll break our comments into the steps that you were asked to do above.

Research

We talked with a friend that works in the publishing industry. We were curious to learn about how the relationships between an author, book, agent, and publishing company work. Many of the questions we asked in our interview were aimed at clarifying the relationships between these entities.

For example, once we established that an agent works with many authors, we asked whether an author would ever work with more than one agent. We wanted to figure out if the agent-to-author relationship was one-to-many or many-to-many. The answer was "sometimes" — an author could sometimes work with one agent for books and another agent for movies. Usually, those two agents would work at the same agency. Questions like this helped us determine how we wanted to set up the database. Ultimately, we decided to restrict our database to not include movies, and as a result, decided that an author could only have one agent.

We asked similar questions about the relationship between books, publishers, and editors — and we learned about the existence of imprints and their relationship to publishers!

Ultimately all of this research led us to the Entity-Relationship diagram included in the downloaded files.

Note that in reality, we would probably have many more fields in all of these tables. After all, having a `publisher` table with only a `name` field isn't very helpful. For the purposes of this exercise, we focused on setting up the skeleton of the relationships between all of these entities. We could have spent much more time adding more details to all of these tables!

Turn Your Diagram Into A Database Schema

The code we used to turn our diagram into a real Postgres database can be found in the `publishing_database.sql` file in the downloads. One thing to note: as we were creating the `age_range` column for the `imprint` table, we thought that it made more sense to split it into `minimum_age` and `maximum_age` columns of type `int`. Making these columns `int`s will allow us to filter the table for something like imprints that publish books for kids under age 20, whereas that would not be possible with a `text` field representing the age range.

Add Data To Your Tables

Check out `publishing_database.sql` to see the INSERT statements we wrote to add data to our tables. We made sure to insert rows that would test the one-to-many and many-to-many relationships that we had previously established. For example, we wanted to make sure that multiple agents could belong to the same agency.

Edit Your Schema As Necessary

As we were adding data to our tables, we realized a few things. First, we realized that there was no way for us to connect a `book` to an `editor`. We could connect a `book` to a `publisher` and an `editor` to a `publisher`, but because multiple editors could work for the same publisher, there was no way to know which editor was editing a book. As a result, we added the `editor_name` column in the `book` table.

Once we added this column, we realized that the `publisher_name` field in the `book` table was redundant. Because each `editor` could only work at one `publisher` (technically, one `imprint` that belongs to a `publisher`), we could find the `publisher` of a `book` through the `editor`. As a result, we removed the `publisher_name` column from the `book` table.

It's debatable whether this is a good idea or not! It might be annoying to have to look up the publisher of a book through its editor every time we're curious about the publisher. Or an editor could switch what publisher they're working for. Ultimately, decisions like that will be dependent upon who ends up using the database. At this point, it might be a good idea to go back to the person that you interviewed and see if there are any obvious flaws in your design! You can see the PostgreSQL code we used to make these edits in `publishing_database.sql`.