In this lesson we are talking about other datatypes those are available for us in Postgres. There are plenty of other datatypes aside from numbers, strings and dates; although these are most commonly used ones.

For example, we can store images in Postgres. There is a special type of data that we can use that allows us to store images up to 1GB in size. Naturally, this is also limited by the computer that is hosting the database. The field that is used to store images can also be used to store other binary files such as anything we can think of really. The datatype is called BYTEA. It is used in Postgres to store bytes, and that is anything we want.

In order to store images, we need to remember how images are made. An image is not just a magical entity as we know, but rather it is composed of pixels. Each of the pixels, each of the small dots that make up the image has a color and some more information associated with it. If we get that information for each of the pixels, we get the color, the transparency and other things, then we can get them all, each of the individual pixels and we can put all that information into a BYTEA field and that is quite a long field but all the characters that compose an image are only a few kilobytes or depending how large our image is it can be up to few megabytes but as we said the field suppose up to 1GB in size so that is something that we can do.

Usually, people recommend that we should store our images into a filesystem outside of the database and then our database can contain a path of where the image is store. For example, we can have our images in our C drive as ‘C:\images\my\_image.jpeg’ and our SQL database can contain this path, our Python program can go into that path and read the image and then show it to the user. That is also another option. Those are just two ways that we can follow to store images.

Another type of data that is quite useful is the ENUM that stands for enumeration. An ENUM is just like a string but it is more than that. It is a set of strings that can limit on what type of data we can put in into a row. For example, it can be used to limit a gender of customers or the possible color of items and so on.

Let’s quickly investigate a sample ENUM and how we can go on using it.

We are creating a new datatype called mood that we can use in our table’s column definitions in parentheses we can put options that would limit the values.

It is creating a new datatype that we can use in our CREATE TABLE command and set only the values mentioned in the parentheses of ENUM. That is quite useful when we are trying to limit of what values can only go in.

CREATE TYPE mood AS ENUM (‘extremely unhappy’, ‘unhappy’, ‘ok’, ‘happy’, ‘extremely happy’);

So, we have created the type and now we will create a table that will carry ENUM values under the *mood* datatype.

CREATE TABLE students (

name character varying(255),

current\_mood mood

);

The query runs successfully, and out table gets created. Now if we select all from our table, we will get output as below,

|  |  |
| --- | --- |
| name  character varying (255) | current\_mood  mood |

As you can see, we have a column called *current mood* of the datatype *mood*. As our table is empty, the output does not show any entry in it. That is why we will INSERT values into this table.

INSERT INTO students VALUES (‘Moe’, ‘happy’);

Our query runs successfully, thus the values get inserted into our table. We will INSERT couple of more values into this table so show that this is quite interesting, especially when we do comparisons and things like that.

INSERT INTO students VALUES (‘Larry’, ‘happy’);

INSERT INTO students VALUES (‘Rolf’, ‘extremely unhappy’);

INSERT INTO students VALUES (‘Jose’, ‘extremely happy’);

INSERT INTO students VALUES (‘Anna’, ‘happy’);

INSERT INTO students VALUES (‘Robert’, ‘unhappy’);

Okay so we have run these queries and they ran successfully.

Now, we will select all from our table,

SELECT \* FROM students;

|  |  |
| --- | --- |
| name  character varying (255) | current\_mood  mood |
| Moe | happy |
| Larry | happy |
| Rolf | extremely unhappy |
| Jose | extremely happy |
| Anna | happy |
| Robert | unhappy |

And we can see that we have got a bit of variety there in case of happiness of our students. We have got three students those are happy, one that is extremely happy, one that is extremely unhappy and one that is unhappy.

Remember when we declared the datatype *mood,* we placed ‘extremely unhappy’ at the beginning, and then ‘unhappy’, then ‘ok’, then ‘happy’, and then ‘extremely happy’. That is important because ENUMs are ordered. The ‘extremely unhappy’ is value number 0 and ‘extremely happy’ is value number 4 which means we can filter by this ENUM.

So, we can write our query as below,

SELECT \* FROM students WHERE current\_mood > ‘ok’;

And that would should give us happy and extremely happy.

SELECT \* FROM students WHERE current\_mood > ‘ok’;

|  |  |
| --- | --- |
| name  character varying (255) | current\_mood  mood |
| Moe | happy |
| Larry | happy |
| Jose | extremely Happy |
| Anna | happy |

And it does, because ‘ok’ is value number 2 and ‘happy’ is 3 and ‘extremely happy’ is 4; 3 and 4 are greater than 2 and that is it. This something we can do with ENUM and they are sometimes useful, maybe not useful to make comparisons like this for genders, but in case happiness or scores and things like that, we may want to limit the inputs and then still be able to perform filters like this one then ENUMs are quite useful. Also, because they limit the inputs which mean we can share responsibility with other people entering data into our database and then also make sure that no incorrect data is input because the ENUM is going to limit that.

Now we will talk about something new in Postgres, and it has been for last couple of versions which is the ability to store JSON, JavaScript Object Notation formatted strings in Postgres. Relational databases have always worked with tables and each table has a bunch of columns and then a bunch of rows and then we can do our filtering or our sorting or things like that as a norm.

JSON is less structured which means we can have a column that has a string of JSON and then a column that has a completely formatted JSON string whereas on the last table all we have are a name and a mood and we cannot have an extra column just for one user. If we add an extra column all our user has that extra column. In JSON some users can have more columns than others and things like that. It is interesting because it gives us a bit more flexibility.

However, it comes for a price, because it is more flexible it means that searching and filtering are slower. If we use a NoSQL database, it specializes in storing JSON formatted strings versus a relational database which specializes in storing tables with related data. But now that Postgres can support JSON then this is something we might consider storing depending upon our applications.