

Full Stack Development BootCamp - Day 3

Build Real-World Apps from Backend to Frontend!

Trainer:

Muhammad Maaz Sheikh

Technical Team Lead



Tech Stack Covered:

- Nest Js
- PostgreSQL
- React Js
- Real world project development

Organized By:

Iqra University

Recap Day 2





Table of Content

Section A-2: Nest Js Advanced Features

- Middleware, Interceptors, Guards, and Pipes
- Error Handling in NestJS (Exception Filters)

Section A-3: PostgreSQL

- Introduction to PostgreSQL
- What is PostgreSQL?
- PostgreSQL History
- Installation Guide & Setup.
- Connect With Databases
- Defining Create, Get, Update & Delete Query.
- Defining Add, Alter, Drop statement.
- PostgreSQL Operators
- JOINS in postgresql
- UNIONS in postgresql
- Group By & Aggregate functions
- PostgreSQL Like Operator
- PostgreSQL In & Not In Operator
- PostgreSQL Order By

Middleware, Interceptors, Guards and Pipes





Middleware, Interceptors, Guards and Pipes

1. Middleware

Purpose:

- Middleware is used for pre-processing requests before they reach the route handlers (controllers).
- It works at the framework level (directly with the HTTP request and response objects).

Execution Order:

- Runs before the controller.

Use Case:

- Logging incoming requests.
- Adding or modifying headers in the request.
- Validating API keys.
- Parsing request data or handling CORS.

Middleware, Interceptors, Guards and Pipes

Middleware...

Key Points:

- Middleware does not handle responses after the controller logic.
- It operates globally or for specific routes.

Example:

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
    next();
  }
}
```

Use:

```
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(LoggerMiddleware).forRoutes('*');
  }
}
```



Middleware, Interceptors, Guards and Pipes

Middleware...

How it Works:

- `@Injectable()`
 - Marks this class as injectable so NestJS can manage it using its Dependency Injection (DI) system.
- implements `NestMiddleware`
 - Ensures this class conforms to the NestJS middleware structure, which must include a `use()` method.
- `use(req, res, next)`
 - This is the middleware function.

Parameters:

req: Incoming HTTP request

res: Response object (Response)

next: Callback to pass control to the next middleware or route (**Without next (), the request would hang and not move to the next middleware or controller.**)



Middleware, Interceptors, Guards and Pipes

2. Guard

Purpose:

- Guards are used for authorization and access control.
- They decide whether a request is allowed to proceed to the controller.

Execution Order:

- Runs **before the controller** (after middleware).

Use Case:

- Restrict access to routes based on roles (e.g., admin-only routes).
- Enforce authentication using tokens.
- Control access to sensitive endpoints.



Middleware, Interceptors, Guards and Pipes

Guard...

Key Points:

- Guards return true (allow access) or false (deny access).
- Can extract and validate user information (e.g., from tokens).

Example:

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    const token = request.headers.authorization;
    return token === 'valid-token';
  }
}
```



Middleware, Interceptors, Guards and Pipes

Guard...

How it Works:

- @Injectable()
 - Marks this class as a provider so Nest can inject it where needed
- implements CanActivate
 - Ensures the class has a canActivate() method.
- This method returns true (allow request) or false (deny request).
- canActivate(context: ExecutionContext): boolean
 - This is the core method.
- **context.switchToHttp().getRequest()** gets the raw request object.
 - It checks for an Authorization header



Middleware, Interceptors, Guards and Pipes

3. Interceptor

Purpose:

- Interceptors are used to wrap the entire request-response lifecycle.
- They allow pre-processing of requests and post-processing of responses.

Execution Order:

- Runs **before and after the controller**.

Use Case:

- Transform or modify incoming requests and outgoing responses.
- Add metadata to responses (e.g., timestamps, pagination info).
- Measure the execution time of requests.

Middleware, Interceptors, Guards and Pipes

Interceptor...

Key Points:

- Can modify both the request and the response.
- Useful for logging, caching, and response formatting.

Example:

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log('Before...');
    return next.handle().pipe(tap(() => console.log('After...')));
  }
}
```



Middleware, Interceptors, Guards and Pipes

Interceptor...

How it Works:

- **@Injectable()**
 - Tells NestJS this class can be injected as a provider.
- Implements NestInterceptor
 - Forces this class to have the intercept() method.

The intercept() Method:

intercept(context: ExecutionContext, next: CallHandler): Observable<any>

Parameters:

- context: Gives access to the execution context (like request/response, route metadata, etc.)
- next: Represents the next step in the request flow. Think of it like: *“Now go to the actual handler.”*



Middleware, Interceptors, Guards and Pipes

Interceptor...

How it Works:

- Inside the Method:

```
console.log('Before...');
```

Runs before the request is handled by the controller/method.

```
return next.handle().pipe(tap(() => console.log('After...')));
```

- **next.handle()** executes the actual route handler.
- **.pipe(tap(...))** runs logic after the route handler returns a value.
- **tap()** comes from rxjs — it's used to peek into the response without modifying it.



Middleware, Interceptors, Guards and Pipes

Consume Interceptor...

Use in Controller:

When Interceptor applied on the controller it will execute on request and response

```
L0  @Controller('user')
L1  @UseInterceptors([LoggingInterceptor])
L2  export class UserController {
L3      constructor(private readonly userService) {}
L4  }
```



Middleware, Interceptors, Guards and Pipes

4. Pipe

Purpose:

- Pipes are used for data **validation** and **transformation**.
- They transform or validate incoming data before it reaches the route handler.

Execution Order:

- Runs **before the controller method** (Parameter-level, route-level, or globally).

Use Case:

- Validate request parameters, query strings, or bodies.
- Convert data types (e.g., string to integer).
- Enforce specific rules for incoming data (e.g., ensuring a field is non-empty).



Middleware, Interceptors, Guards and Pipes

Pipe...

Key Points:

- Works at the parameter or route level.
- Returns transformed data or throws an error if validation fails.

Example:

```
@Injectable()
export class ParseIntPipe implements PipeTransform {
  transform(value: string): number {
    const parsedValue = parseInt(value, 10);
    if (isNaN(parsedValue)) {
      throw new BadRequestException('Validation failed');
    }
    return parsedValue;
  }
}
```



Middleware, Interceptors, Guards and Pipes

Pipe...

How it Works:

- The `transform()` Method:

```
transform(value: any)
```

Receives the raw input (like query params, route params, body, etc.), and does two things:

1. Transforms it into an integer using `parseInt()`
2. Validates that the result is indeed a number

Steps:

- `const val = parseInt(value, 10);`
- Converts string input to an integer (base 10)
- Throws a 400 Bad Request if input isn't a valid number
- If valid, returns the number for the controller to use

Middleware, Interceptors, Guards and Pipes

Consume Pipes...

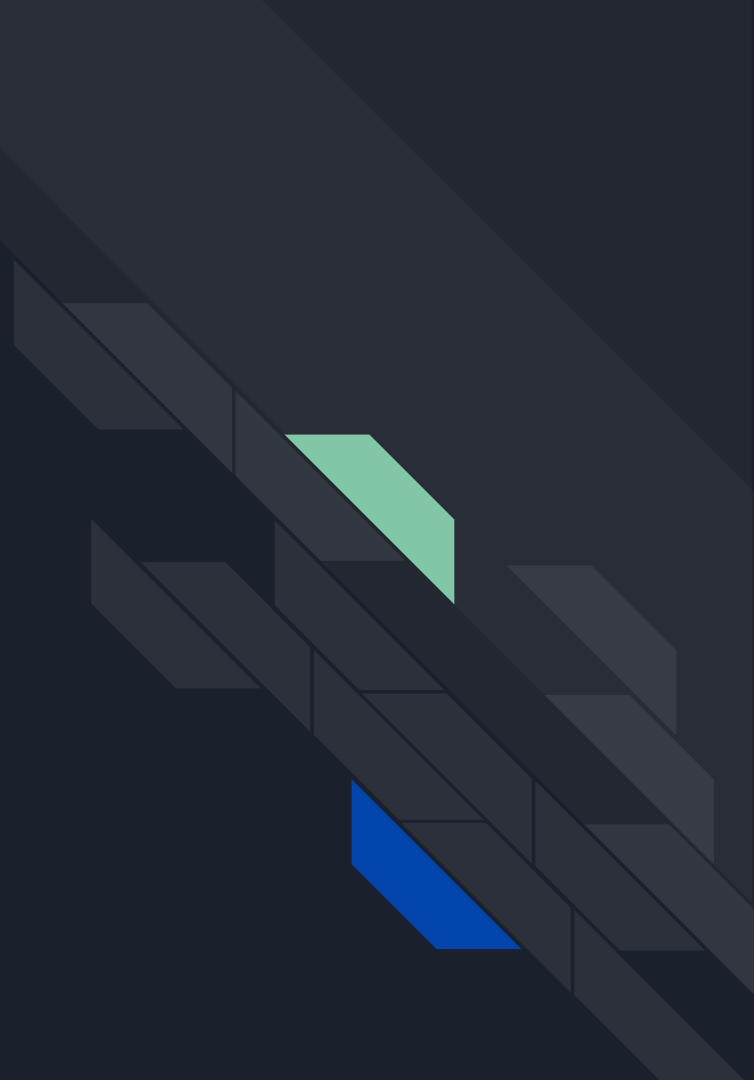
Use in Controller:

When pipes created you can consume pipes in your controller and action method as define below

```
@Controller('user')
@UseInterceptors(LoggingInterceptor)
@UsePipes(ParseIntPipe)
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Get('/:id')
  @UseGuards(AuthGuard)
  getUser(@Param('id') id: ParseIntPipe) {
    return this.userService.getUser(Number(id));
  }
}
```

Error Handling in Nest JS





Error Handling in Nest JS

Introduction

Error handling is a crucial aspect of any application development, ensuring that the application remains robust and resilient in the face of unexpected situations. In this section, we'll delve into various techniques and best practices for implementing better error handling in Nest.js applications.

Understanding Error Handling in Nest.js

What is Error Handling?

Error handling refers to the process of anticipating, detecting, and resolving errors that occur during the execution of a program. In the context of Nest.js, errors can arise from various sources, including user input validation failures, database queries, external API calls, and unexpected runtime exceptions.



Error Handling in Nest JS

Why is Effective Error Handling Important?

Effective error handling is critical for several reasons:

1. **Improved User Experience:** Proper error handling ensures that users receive informative and actionable feedback when something goes wrong, enhancing the overall user experience.
2. **Debugging and Maintenance:** Well-handled errors make it easier to identify and diagnose issues during the development and maintenance phases.
3. **Security:** Properly handling errors can prevent information leakage and mitigate security vulnerabilities by not exposing sensitive information to users or attackers.



Error Handling in Nest JS

Implementing Better Error Handling in Nest JS

Now, let's explore some best practices and techniques for improving error handling in Nest.js applications:

1. Use Custom Exception Filters

Custom exception filters allow you to intercept specific types of exceptions and customize the error response sent back to the client. By creating custom exception filters, you can tailor error messages, status codes, and response formats to meet the requirements of your application.

```
import { Catch, ExceptionFilter, ArgumentsHost, HttpException } from '@nestjs/common';
import { Response } from 'express';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const status = exception.getStatus();

    response
      .status(status)
      .json({
        statusCode: status,
        message: exception.message,
      });
  }
}
```

Error Handling in Nest JS

2. Implement Global Exception Handling

Global exception handling allows you to centralize error handling logic and gracefully handle uncaught exceptions that occur during request processing. By implementing global exception filters, you can ensure consistent error handling across all parts of your application.

```
import { ExceptionFilter, Catch, ArgumentsHost } from '@nestjs/common';

@Catch()
export class GlobalExceptionHandler implements ExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const status = exception instanceof HttpException ? exception.getStatus() :
    response.status(status).json({
      statusCode: status,
      message: 'Internal server error',
    });
  }
}
```




Error Handling in Nest JS

3. Use Validation Pipes for Input Validation

Nest.js provides built-in validation pipes that can automatically validate incoming request payloads against predefined validation rules. By using validation pipes, you can ensure that only valid data is processed by your application, reducing the likelihood of runtime errors due to malformed input.

```
import { ValidationPipe } from '@nestjs/common';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
bootstrap();
```

Error Handling in Nest JS

4. Handle Database Errors Gracefully

When working with databases in Nest.js applications, it's essential to handle database-related errors gracefully. You can use try-catch blocks or asynchronous error-handling techniques to catch and handle database errors effectively.

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './user.entity';

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(User)
    private readonly userRepository: Repository<User>,
  ) {}

  async createUser(user: User): Promise<User> {
    try {
      return await this.userRepository.save(user);
    } catch (error) {
      throw new Error('Failed to create user');
    }
  }
}
```

SQL & PostgreSQL





Introduction to SQL

What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987



Introduction to SQL

What is RDBMS?

- RDBMS stands for Relational Database Management System.
- RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.
- The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.



Introduction to PostgreSQL

What is PostgreSQL?

- PostgreSQL is a free open-source database system that supports both relational (SQL) and non-relational (JSON) queries.
- PostgreSQL is a back-end database for dynamic websites and web applications.
- PostgreSQL supports the most important programming languages:
 - ◆ Python
 - ◆ Java
 - ◆ C/C++
 - ◆ C#
 - ◆ Node.js
 - ◆ Go
 - ◆ Ruby



Introduction to PostgreSQL

PostgreSQL History

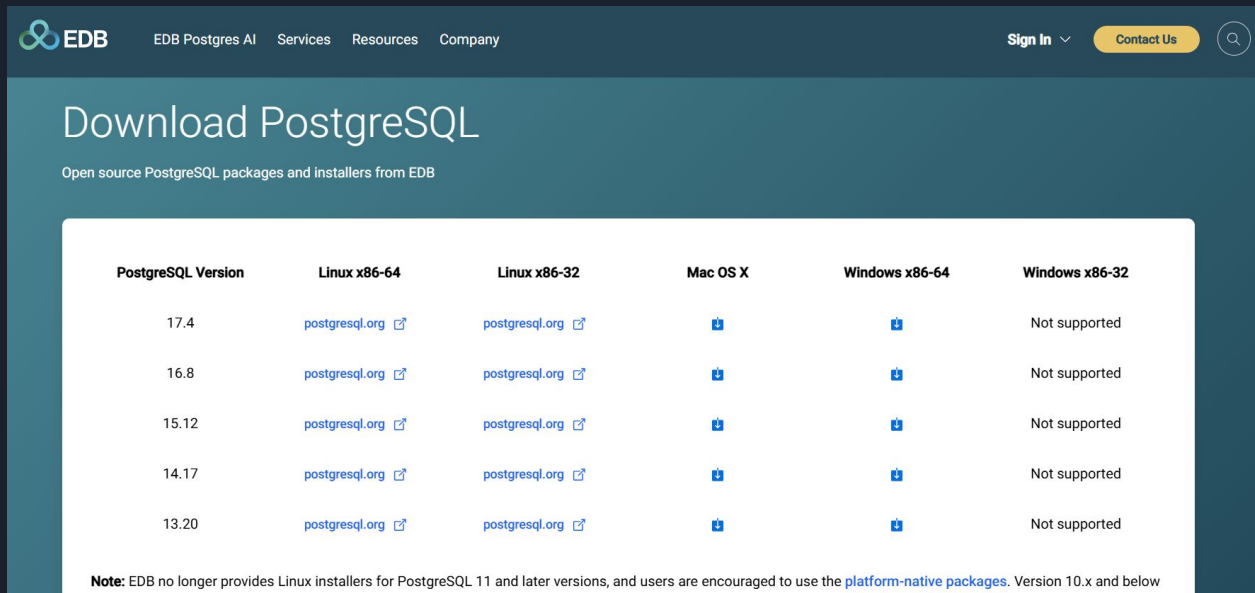
- PostgreSQL was invented at the Berkeley Computer Science Department, University of California.
- It started as a project in 1986 with the goal of creating a database system with the minimal features needed to support multiple data types.
- In the beginning, PostgreSQL ran on UNIX platforms, but now it can run on various platforms, including Windows and MacOS.











Installation Guide

Download PostgreSQL

To install PostgreSQL locally on your computer, visit the [installer by EDB](#), and download the newest version compatible with your operating system.

I will choose the newest Windows version:

The screenshot shows the EDB PostgreSQL download page. The header includes the EDB logo and navigation links for EDB Postgres AI, Services, Resources, and Company. There are also links for Sign In and Contact Us. The main heading is "Download PostgreSQL" with a subtitle "Open source PostgreSQL packages and installers from EDB". Below this is a table listing PostgreSQL versions and their availability for different operating systems. The table has columns for PostgreSQL Version, Linux x86-64, Linux x86-32, Mac OS X, Windows x86-64, and Windows x86-32. The versions listed are 17.4, 16.8, 15.12, 14.17, and 13.20. For each version, the Linux and Mac OS X columns provide links to the PostgreSQL website, while the Windows columns provide download icons. The Windows x86-32 column indicates that these versions are not supported. A note at the bottom states that EDB no longer provides Linux installers for PostgreSQL 11 and later versions, and users are encouraged to use platform-native packages for version 10.x and below.

PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
17.4	postgresql.org	postgresql.org			Not supported
16.8	postgresql.org	postgresql.org			Not supported
15.12	postgresql.org	postgresql.org			Not supported
14.17	postgresql.org	postgresql.org			Not supported
13.20	postgresql.org	postgresql.org			Not supported

Note: EDB no longer provides Linux installers for PostgreSQL 11 and later versions, and users are encouraged to use the [platform-native packages](#). Version 10.x and below



Installation Guide

Installation Steps:

1. When the downloading is complete, double click the downloaded file and start the installation
2. You can specify the location of PostgreSQL, I will go with the default choice
3. To use PostgreSQL, you will need to install the PostgreSQL Server. In this tutorial we will also use the pgAdmin 4 component, and the Command Line Tools
4. You can also choose where to store the database data, I will go with the default choice
5. You will have to select a password to get access to the database. Since this is a local database, with no incoming connection, I will choose the password **12345678**
6. You can set the port the server should listen on, I will go with the default choice which is **5432**.
7. Select the geographical location of the database server
8. If everything looks OK, click '**Next**' to continue



PostgreSQL Get Started

Connect to the Database:

If you have followed the steps from the Install PostgreSQL page, you now have a PostgreSQL database on your computer.

There are several ways to connect to the database, we will look at two ways in this tutorial:

- pgAdmin 4

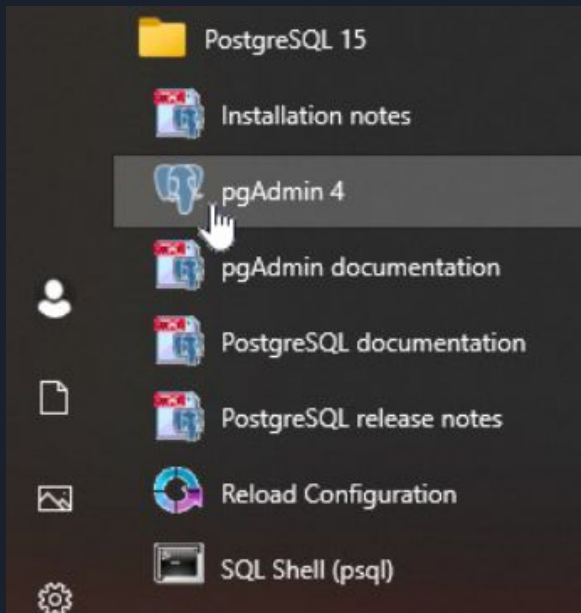
Both of them come with the installation of PostgreSQL

PostgreSQL Get Started

Start pgAdmin4

You will find the pgAdmin4 application in the start menu under PostgreSQL:

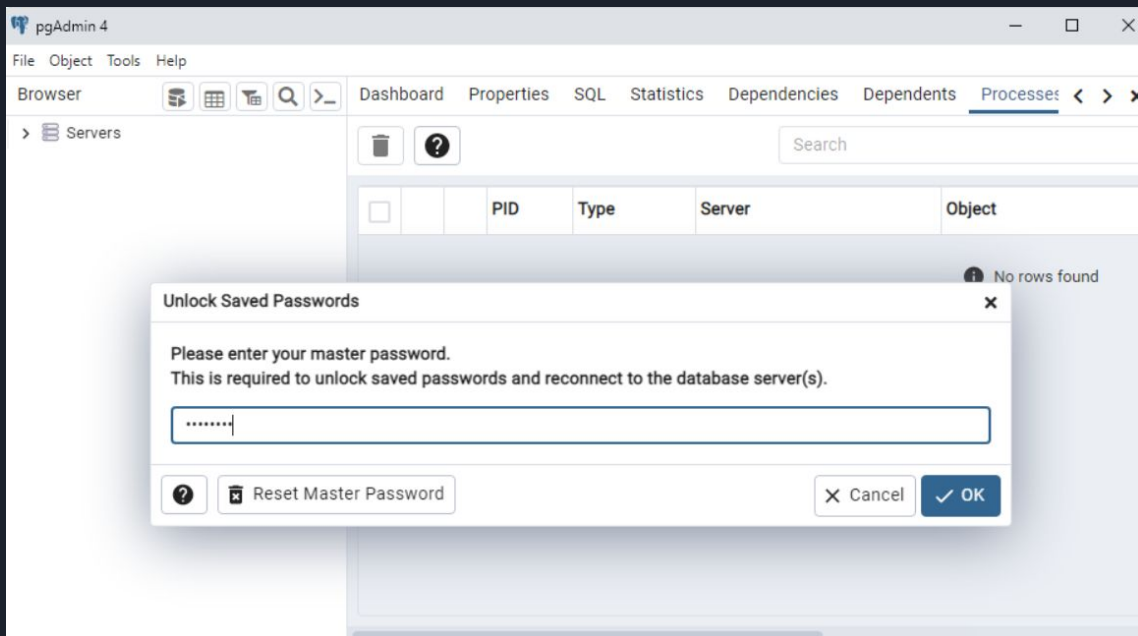
Tip: If you cannot find it, try searching for "**pgAdmin4**" on your computer.



PostgreSQL Get Started

Start pgAdmin4

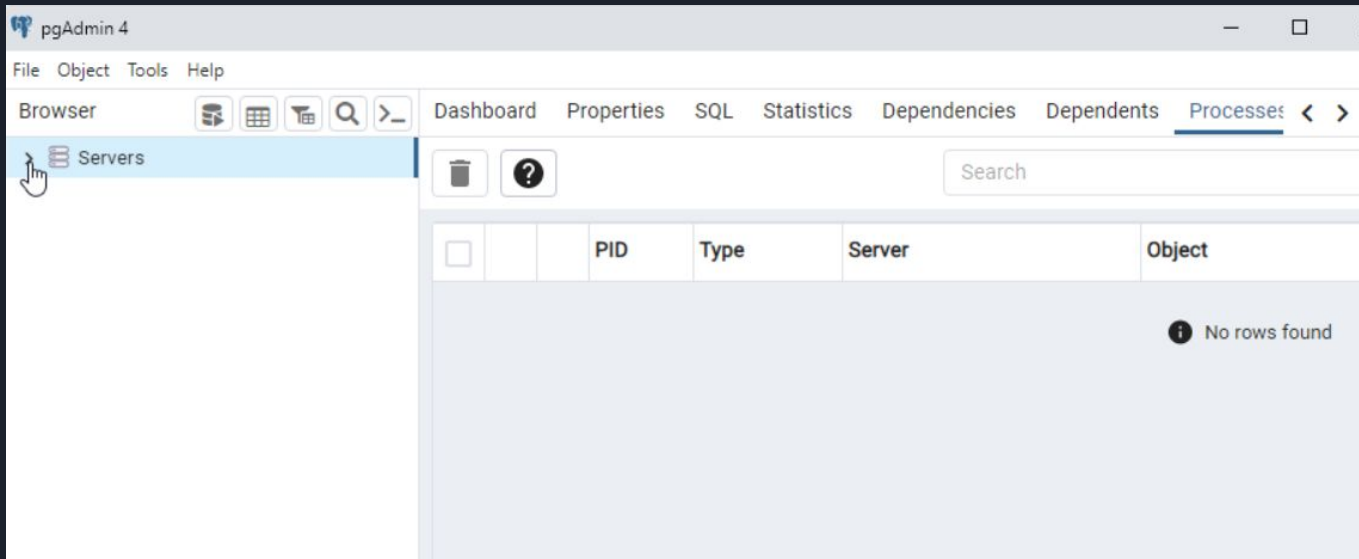
Once the program has started, you should see a window like the one below, choose a master password, Since this is a local database that will run only on my computer, I will once again choose the password 12345678:



PostgreSQL Get Started

pgAdmin4

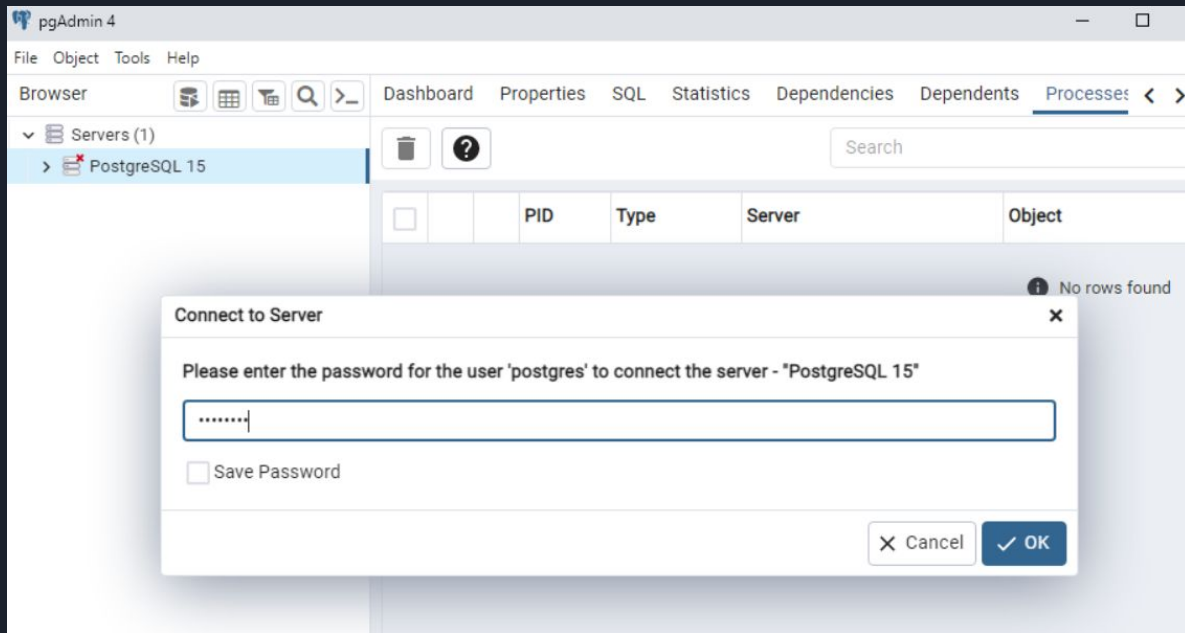
- Once you are inside the program, try to perform a simple SQL query.
- To do that we have to navigate to the database.
- Start by opening the [Servers] option in the menu on the left:



PostgreSQL Get Started

Connect to Server

Now you need to enter the password that you created when you installed PostgreSQL, my password is **12345678**:





PostgreSQL Get Started

Find Database

- Click on the [Database] option on in the menu on the left
- You should find a database named postgres, right-click it choose the "Query Tool"
- In the Query Tool we can start executing SQL statements.
- Our database is empty, so we cannot query any tables yet, but we can check the version with this SQL statement:

```
SELECT version();
```

PostgreSQL Create Table

Create Table

The following SQL statement will create a table named cars in your PostgreSQL database:

Query	Query History
1	CREATE TABLE cars (
2	brand VARCHAR (255),
3	model VARCHAR (255),
4	year INT
5);

SQL Statement Explained

The above SQL statement created an empty table with three fields: **brand**, **model**, and **year**.

When creating fields in a table we have to specify the data type of each field.

For brand and model we are expecting string values, and string values are specified with the **VARCHAR** keyword.

We also have to specify the number of characters allowed in a string field, and since we do not know exactly, we just set it to 255.

For **year** we are expecting integer values (numbers without decimals), and integer values are specified with the **INT** keyword.

PostgreSQL Create Table

Display Table

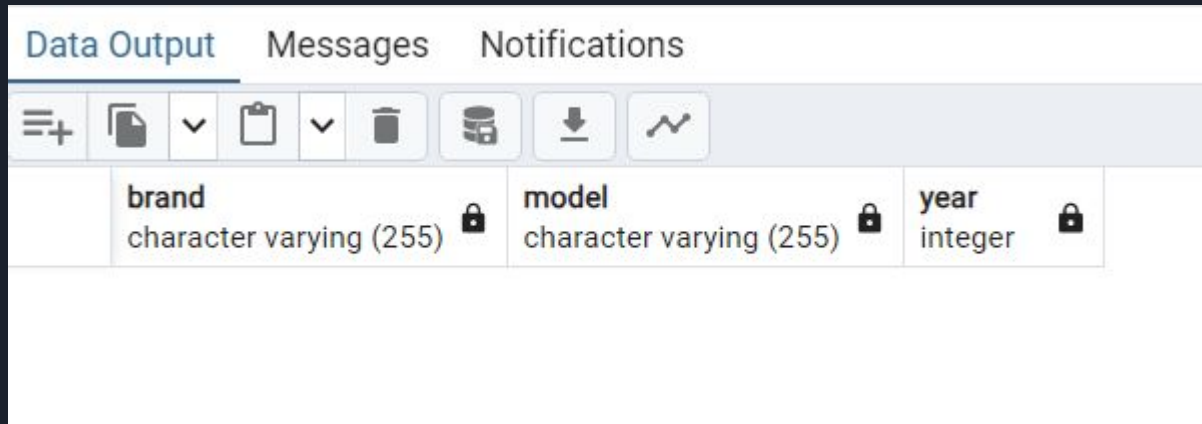
You can "display" the empty table you just created with another SQL statement:



Query Query History

```
1 SELECT * FROM cars;  
2
```

Which will give you this result as empty rows:



Data Output		Messages	Notifications
<div>Icons for table operations: expand, save, dropdown, clipboard, dropdown, delete, database, download, chart</div>			
	brand character varying (255) 🔒	model character varying (255) 🔒	year integer 🔒



PostgreSQL Insert Data

Insert Into

To insert data into a table in PostgreSQL, we use the **INSERT INTO** statement.

The following SQL statement will insert one row of data into the cars table you created in the previous slide.

Query	Query History
1	INSERT INTO cars (brand, model, year)
2	VALUES ('Ford', 'Mustang', 1964);
3	

SQL Statement Explained

As you can see in the SQL statement above, string values must be written with apostrophes.

Numeric values can be written without apostrophes, but you can include them if you want.



PostgreSQL Fetch Data

Select Data

To retrieve data from a database, we use the **SELECT** statement.

Specify Columns

By specifying the column names, we can choose which columns to select:

Example: `SELECT brand, year FROM cars;`

Return All Columns

Specify a ***** instead of the column names to select **all** columns:

Example: `SELECT * FROM cars;`

PostgreSQL Update Data

Update Statement

The **UPDATE** statement is used to modify the value(s) in existing records in a table.

Example: Set the color of the Volvo to 'red':

Query	Query History
1	UPDATE cars
2	SET color = 'red'
3	WHERE brand = 'Volvo';

--->

Result

UPDATE 1

Which means that **1** row was affected by the **UPDATE** statement.

Note:

- Be careful with the **WHERE** clause, in the example above **ALL** rows where brand = 'Volvo' gets updated.
- Be careful when updating records. If you omit the **WHERE** clause, **ALL** records will be updated!



PostgreSQL Update Data

Update Multiple Columns

To update more than one column, separate the name/value pairs with a comma ,;

Example: Update color and year for the Toyota:

Query	Query History
1	<code>UPDATE cars</code>
2	<code>SET color = 'white', year = 1970</code>
3	<code>WHERE brand = 'Toyota';</code>

--->

Result

UPDATE 1



PostgreSQL Delete Data

The Delete Statement

The **DELETE** statement is used to delete existing records in a table.

Example: Delete all records where **brand** is 'Volvo':

```
DELETE FROM cars  
WHERE brand = 'Volvo';
```

--->

```
DELETE 1
```



PostgreSQL Add Column

Alter table Statement

- To add a column to an existing table, we have to use the ALTER TABLE statement.
- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

Add Column

We want to add a column named color to our cars table.

When adding columns we must also specify the data type of the column. Our color column will be a string, and we specify string types with the VARCHAR keyword. we also want to restrict the number of characters to 255:

Example: Add a column named **color**:

:

```
ALTER TABLE cars  
ADD color VARCHAR(255);
```



PostgreSQL Alter Column

Alter table Statement

- To change the data type, or the size of a table column we have to use the ALTER TABLE statement.
- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

Alter Column

We want to change the data type of the year column of the cars table from INT to VARCHAR(4).

To modify a column, use the ALTER COLUMN statement and the TYPE keyword followed by the new data type:

Example: Change the year column from INT to VARCHAR(4):

```
ALTER TABLE cars  
ALTER COLUMN year TYPE VARCHAR(4);
```




PostgreSQL Drop Column

Alter table Statement

- To remove a column from a table, we have to use the ALTER TABLE statement.
- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

Drop Column

We want to remove the column named color from the cars table.

To remove a column, use the DROP COLUMN statement:

Example: Remove the color column:

```
ALTER TABLE cars  
DROP COLUMN color;
```



PostgreSQL Drop Column

Alter table Statement

- To remove a column from a table, we have to use the ALTER TABLE statement.
- The ALTER TABLE statement is used to add, delete, or modify columns in an existing table.
- The ALTER TABLE statement is also used to add and drop various constraints on an existing table.

Drop Column

We want to remove the column named color from the cars table.

To remove a column, use the DROP COLUMN statement:

Example: Remove the color column:

```
ALTER TABLE cars  
DROP COLUMN color;
```



PostgreSQL Drop Table

Drop table Statement

The DROP TABLE statement is used to drop an existing table in a database.

Drop Column

The following SQL statement drops the existing table cars:

Example: Delete the cars table:

```
DROP TABLE cars;
```



PostgreSQL Operators

Arithmetic Operators

Operator	Description	Example
+	Addition	<code>5 + 2 = 7</code>
-	Subtraction	<code>5 - 2 = 3</code>
*	Multiplication	<code>5 * 2 = 10</code>
/	Division	<code>5 / 2 = 2.5</code>
%	Modulus (remainder)	<code>5 % 2 = 1</code>



PostgreSQL Operators

Comparison Operators

Operator	Description	Example
=	Equal	<code>salary = 5000</code>
!= or <>	Not equal	<code>role != 'HR'</code>
<	Less than	<code>age < 30</code>
>	Greater than	<code>age > 30</code>
<=	Less than or equal	<code>age <= 30</code>
>=	Greater than or equal	<code>age >= 30</code>



PostgreSQL Operators

Logical Operators

Operator	Description	Example
AND	All conditions must be true	<code>age > 20 AND salary < 6000</code>
OR	At least one condition is true	<code>role = 'HR' OR role = 'Admin'</code>
NOT	Negates a condition	<code>NOT (salary > 5000)</code>

Like and ILike

Operator	Description	Example
LIKE	Pattern match (case-sensitive)	<code>name LIKE 'A%'</code>
ILIKE	Case-insensitive LIKE	<code>name ILIKE 'a%'</code>



PostgreSQL Operators

In and Between

In: Match any value in a list

sql

```
SELECT * FROM users WHERE role IN ('Admin', 'HR');
```

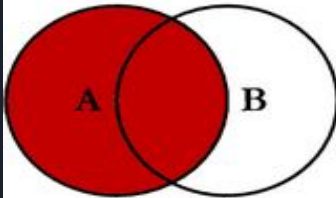
Between: Check range (inclusive)

sql

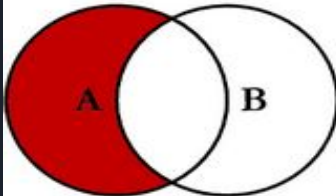
```
SELECT * FROM products WHERE price BETWEEN 100 AND 500;
```

Joins in PostgreSQL

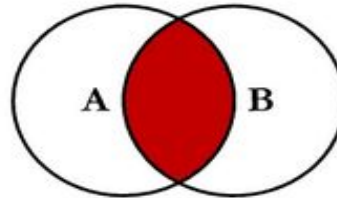
SQL JOINS



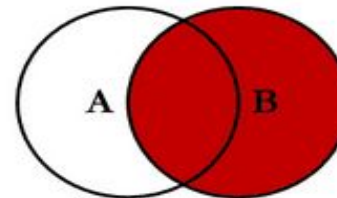
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



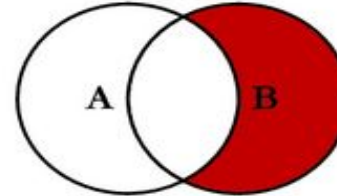
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



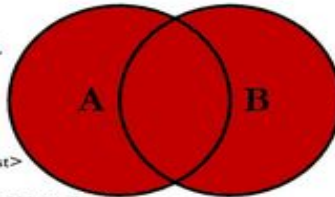
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



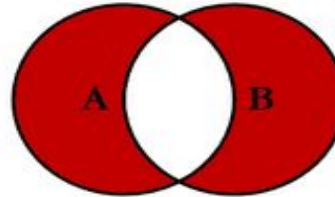
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```


Joins in PostgreSQL

1. INNER JOIN

The **INNER JOIN** keyword selects records that have matching values in both tables.

Let's look at an example:

testproducts:

testproduct_id	product_name	category_id
1	Johns Fruit Cake	3
2	Marys Healthy Mix	9
3	Peters Scary Stuff	10
4	Jims Secret Recipe	11
5	Elisabeths Best Apples	12
6	Janes Favorite Cheese	4
7	Billys Home Made Pizza	13
8	Ellas Special Salmon	8
9	Roberts Rich Spaghetti	5
10	Mias Popular Ice	14

(10 rows)

categories:

category_id	category_name
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

(8 rows)



Joins in PostgreSQL

1. INNER JOIN

Notice that many of the products in testproducts have a category_id that does not match any of the categories in the categories table.

By using INNER JOIN we will not get the records where there is not a match, we will only get the records that matches both tables:

```
SELECT testproduct_id, product_name, category_name
FROM testproducts
INNER JOIN categories ON testproducts.category_id = categories.category_id;
```

Result:

Only the records with a match in BOTH tables are returned:

testproduct_id	product_name	category_name
1	Johns Fruit Cake	Confections
6	Janes Favorite Cheese	Dairy Products
8	Ellas Special Salmon	Seafood
9	Roberts Rich Spaghetti	Grains/Cereals

(4 rows)

Joins in PostgreSQL

2. LEFT JOIN

The **LEFT JOIN** keyword selects ALL records from the "left" table, and the matching records from the "right" table. The result is 0 records from the right side if there is no match.

Let's look at an example:

testproducts:

testproduct_id	product_name	category_id
1	Johns Fruit Cake	3
2	Marys Healthy Mix	9
3	Peters Scary Stuff	10
4	Jims Secret Recipe	11
5	Elisabeths Best Apples	12
6	Janes Favorite Cheese	4
7	Billys Home Made Pizza	13
8	Ellas Special Salmon	8
9	Roberts Rich Spaghetti	5
10	Mias Popular Ice	14

(10 rows)

categories:

category_id	category_name
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

(8 rows)

Joins in PostgreSQL

2. LEFT JOIN

Many of the products in testproducts have a category_id that does not match any of the categories in the categories table.

By using LEFT JOIN we will get all records from testproducts, even the ones with no match in the categories table:

```
SELECT testproduct_id, product_name, category_name
FROM testproducts
LEFT JOIN categories ON testproducts.category_id = categories.category_id;
```

Result:

All records from

testproducts, and only the matched

records from categories:

testproduct_id	product_name	category_name
1	Johns Fruit Cake	Confections
2	Marys Healthy Mix	
3	Peters Scary Stuff	
4	Jims Secret Recipe	
5	Elisabeths Best Apples	
6	Janes Favorite Cheese	Dairy Products
7	Billys Home Made Pizza	
8	Ellas Special Salmon	Seafood
9	Roberts Rich Spaghetti	Grains/Cereals
10	Mias Popular Ice	

(10 rows)

Joins in PostgreSQL

3. RIGHT JOIN

The **RIGHT JOIN** keyword selects ALL records from the "right" table, and the matching records from the "left" table. The result is 0 records from the left side if there is no match.

Let's look at an example:

testproducts:

testproduct_id	product_name	category_id
1	Johns Fruit Cake	3
2	Marys Healthy Mix	9
3	Peters Scary Stuff	10
4	Jims Secret Recipe	11
5	Elisabeths Best Apples	12
6	Janes Favorite Cheese	4
7	Billys Home Made Pizza	13
8	Ellas Special Salmon	8
9	Roberts Rich Spaghetti	5
10	Mias Popular Ice	14

(10 rows)

categories:

category_id	category_name
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

(8 rows)

Joins in PostgreSQL

3. RIGHT JOIN

Many of the products in testproducts have a category_id that does not match any of the categories in the categories table.

By using RIGHT JOIN we will get all records from categories, even the ones with no match in the testproducts table:

```
SELECT testproduct_id, product_name, category_name
FROM testproducts
RIGHT JOIN categories ON testproducts.category_id = categories.category_id;
```

Result:

All records from **categories**, and only the
matched records from **testproducts**:

testproduct_id	product_name	category_name
1	Johns Fruit Cake	Confections
6	Janes Favorite Cheese	Dairy Products
8	Ellas Special Salmon	Seafood
9	Roberts Rich Spaghetti	Grains/Cereals
		Condiments
		Meat/Poultry
		Beverages
		Produce

(8 rows)

Joins in PostgreSQL

4. FULL JOIN

The **FULL JOIN** keyword selects ALL records from both tables, even if there is not a match. For rows with a match the values from both tables are available, if there is not a match the empty fields will get the value NULL.

Let's look at an example:

testproducts:

testproduct_id	product_name	category_id
1	Johns Fruit Cake	3
2	Marys Healthy Mix	9
3	Peters Scary Stuff	10
4	Jims Secret Recipe	11
5	Elisabeths Best Apples	12
6	Janes Favorite Cheese	4
7	Billys Home Made Pizza	13
8	Ellas Special Salmon	8
9	Roberts Rich Spaghetti	5
10	Mias Popular Ice	14

(10 rows)

categories:

category_id	category_name
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood

(8 rows)

Joins in PostgreSQL

4. FULL JOIN

Many of the **products** in testproducts have a category_id that does not match any of the categories in the categories table.

By using **FULL JOIN** we will get all records from both the **categories** table and the testproducts table:

```
SELECT testproduct_id, product_name, category_name
FROM testproducts
FULL JOIN categories ON testproducts.category_id = categories.category_id;
```

Result:

- All records from **both tables** are returned
- Rows with no match will get a NULL

value in fields from the opposite table:

testproduct_id	product_name	category_name
1	Johns Fruit Cake	Confections
2	Marys Healthy Mix	
3	Peters Scary Stuff	
4	Jims Secret Recipe	
5	Elisabeths Best Apples	
6	Janes Favorite Cheese	Dairy Products
7	Billys Home Made Pizza	
8	Ellas Special Salmon	Seafood
9	Roberts Rich Spaghetti	Grains/Cereals
10	Mias Popular Ice	
		Condiments
		Meat/Poultry
		Beverages
		Produce

(14 rows)



Joins in PostgreSQL

5. CROSS JOIN

The CROSS JOIN keyword matches ALL records from the "left" table with EACH record from the "right" table.

That means that all records from the "right" table will be returned for each record in the "left" table.

This way of joining can potentially return very large table, and you should not use it if you do not have to.

Example:

```
SELECT testproduct_id, product_name, category_name
FROM testproducts
CROSS JOIN categories;
```



Unions in PostgreSQL

UNION

The **UNION** operator is used to combine the result-set of two or more queries.

The queries in the union must follow these rules:

- They must have the same number of columns
- The columns must have the same data types
- The columns must be in the same order

Example:

Combine products and **testproducts** using the **UNION** operator:

```
SELECT product_id, product_name
FROM products
UNION
SELECT testproduct_id, product_name
FROM testproducts
ORDER BY product_id;
```



Unions in PostgreSQL

UNION vs UNION ALL

With the **UNION** operator, if some rows in the two queries returns the exact same result, only one row will be listed, because **UNION** selects only distinct values.

Use **UNION ALL** to return duplicate values.

Example - UNION:

Example - UNION ALL:

Let's make some changes to the queries, so that we have duplicate values in the result:

```
SELECT product_id
FROM products
UNION
SELECT testproduct_id
FROM testproducts
ORDER BY product_id;
```

```
SELECT product_id
FROM products
UNION ALL
SELECT testproduct_id
FROM testproducts
ORDER BY product_id;
```



GROUP BY and Aggregates

GROUP BY

The **GROUP BY** clause groups rows that have the same values into summary rows, like "find the number of customers in each country".

The **GROUP BY** clause is often used with aggregate functions like **COUNT()**, **MAX()**, **MIN()**, **SUM()**, **AVG()** to group the result-set by one or more columns.

Example:

Lists the number of customers in each country:

```
SELECT COUNT(customer_id), country
FROM customers
GROUP BY country;
```



GROUP BY and Aggregates

Aggregates - Sum()

The **SUM()** function returns the total sum of a numeric column.

The following SQL statement finds the sum of the quantity fields in the **order_details** table:

Example:

Return the total amount of ordered items:

```
SELECT SUM(quantity)
FROM order_details;
```

Similarly other aggregates function work - Need to explore by yourself in your practice session

AVG(), MAX(), MIN(),



PostgreSQL LIKE Operator

LIKE

The **LIKE** operator is used in a **WHERE** clause to search for a specified pattern in a column.

There are two wildcards often used in conjunction with the **LIKE** operator:

- % The percent sign represents zero, one, or multiple characters
- _ The underscore sign represents one, single character

Starts with

To return records that starts with a specific letter or phrase, add the % at the end of the letter or phrase.

Example:

Return all customers with a name that starts with the letter 'A':

```
SELECT * FROM customers
WHERE customer_name LIKE 'A%';
```



PostgreSQL LIKE Operator

Contains

To return records that contains a specific letter or phrase, add the % both before and after the letter or phrase.

Example:

Return all customers with a name that contains the letter 'A':

```
SELECT * FROM customers
WHERE customer_name LIKE '%A%';
```

ILIKE

The **LIKE** operator is case sensitive, if you want to do a case insensitive search, use the **ILIKE** operator instead.

Example:

Return all customers with a name that contains the letter 'A' or 'a':

```
SELECT * FROM customers
WHERE customer_name ILIKE '%A%';
```



PostgreSQL LIKE Operator

Ends With

To return records that ends with a specific letter or phrase, add the % before the letter or phrase.

Example:

Return all customers with a name that ends with the phrase 'en':

```
SELECT * FROM customers
WHERE customer_name LIKE '%en';
```

The Underscore _ Wildcard

The _ wildcard represents a single character.

It can be any character or number, but each _ represents one, and only one, character.

Example:

Return all customers from a city that starts with 'L' followed by one wildcard character, then 'nd' and then two wildcard characters:

```
SELECT * FROM customers
WHERE city LIKE 'L_nd__';
```




PostgreSQL IN Operator

IN

The IN operator allows you to specify a list of possible values in the WHERE clause.

The IN operator is a shorthand for multiple OR conditions.

Example:

Return all customers from 'Germany', 'France' or 'UK':

```
SELECT * FROM customers
WHERE country IN ('Germany', 'France', 'UK');
```

NOT IN

By using the NOT keyword in front of the IN operator, you return all records that are NOT any of the values in the list.

Example:

Return all customers that are NOT from 'Germany', 'France' or 'UK':

```
SELECT * FROM customers
WHERE country NOT IN ('Germany', 'France', 'UK');
```



PostgreSQL ORDER BY

Sort Data

The ORDER BY keyword is used to sort the result in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

Example:

Sort the table by price:

```
SELECT * FROM products  
ORDER BY price;
```

DESC

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

Example:

Sort the table by price, in descending order:

```
SELECT * FROM products  
ORDER BY price DESC;
```