Basic Of Java

Package Statement:-The package statement should be at the very top of each file, defining the namespace of your class.

Example:-
```
package com.example.application;
          class{

           }
```

Import Statements:-After the package statement, you add import statements for all the classes and libraries needed for the functionality.

Example:-
```
import java.util.List;
          Class{
         p.s.v.m(){
           List li=new ArrayList();
             }
            }
```

Class Definition:-This defines a class, which can be a main class.

Example;-
```
class{

}
```

Main Method:-The entry point of the application. The main method is typically found in the main application class.

Example:-
```
public static void main(String[] args) {
               SpringApplication.run(Application.class, args);
        }
```

Model Class/Entity class:- This is where you define your data model classes with properties, constructors, getters, setters, and any necessary annotations.

Example:-
```
@Entity
 public class User {
@Id @GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id;
 private String username;
private String password;
 // Constructors, getters, and setters
 }
```

Repositories:-Repository interfaces are used to handle database operations. Spring Data JPA will automatically implement these based on the methods you define.

Example:
```
public interface UserRepository extends JpaRepository<User, Long> {
User findByUsername(String username); //user-defined method to search user by username
}
```

Annotation:-Annotations are used throughout Java classes to provide metadata and configure behaviors.

- `@`***`SpringBootApplication`*** - Indicates a Spring Boot application entry point.

- `@Entity` - Defines a class as an entity (for ORM purposes).
- **`@Id`, `@GeneratedValue`** - Used in model classes to mark the primary key and define generation strategy.
- **`@Repository`** - Marks a class as a repository.
- `@Autowired` - Used for dependency injection, allowing Spring to manage class dependencies.
- **`@RequestMapping`** - Maps HTTP requests to handler methods in controllers.

- **`@Component`**
  - Marks a class as a Spring component. It is a generic stereotype for any Spring-managed component. Spring will autodetect these classes for dependency injection.

- **`@Service`**
  - A specialization of `@Component`, used to annotate classes that provide business logic or services.

- **`@Controller`**
  - Marks a class as a Spring MVC controller, allowing it to handle HTTP requests.

- **`@RestController`**
  - A combination of `@Controller` and **`@ResponseBody`**, used for RESTful web services. It automatically serializes return values to JSON or XML.

@ResponseBody:- When you use `@ResponseBody` on a method, Spring will:

1. Serialize the returned object to JSON (or XML, depending on the configuration).

Loops

While Loop:- A `while` loop repeatedly executes a block of code as long as a given condition is true. It checks the condition before entering the loop body, so if the condition is false initially, the loop will not execute.

```
int i = 1;
while (i <= 5) {
    System.out.println("Count: " + i);
    i++;
}
o/p:-
Count: 1
Count: 2
```

```
Count: 3
Count: 4
Count: 5
```

For Loop:- A `for` loop is typically used when the number of iterations is known beforehand. It consists of three parts in the header: initialization, condition, and update.

```
for (int i = 1; i <= 5; i++) {
    System.out.println("Count: " + i);
}
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Do-While Loop:- A `do-while` loop is similar to a `while` loop, except that it checks the condition *after* executing the loop body. This guarantees that the loop body runs at least once, even if the condition is initially false.

```
int i = 1;
do {
    System.out.println("Count: " + i);
    i++;
} while (i <= 5);
```

Output:

```
Count: 1
Count: 2
Count: 3
Count: 4
Count: 5
```

Nested For Loop:- A nested `for` loop is a `for` loop inside another `for` loop. It is often used to work with multidimensional data structures, like matrices.

```
for (int i = 1; i <= 3; i++) {       // Outer loop
    for (int j = 1; j <= 3; j++) {   // Inner loop
        System.out.print(i * j + " ");
    }
    System.out.println(); // Moves to the next line after each inner
loop
}
```

Output:

```
1 2 3
2 4 6
3 6 9
```

Conditional Statements

Simpile IF Statement:- A simple `if` statement executes a block of code if a condition is true. If the condition is false, the code block is skipped.

```
int number = 10;
if (number > 5) {
    System.out.println("The number is greater than 5.");
}
```

Output (if `number` is 10):

```
The number is greater than 5.
```

IF-Else Statement:- An `if-else` statement allows you to execute one block of code if a condition is true and another block if it is false.

```
int number = 3;
if (number > 5) {
    System.out.println("The number is greater than 5.");
} else {
    System.out.println("The number is 5 or less.");
}
```

Output (if `number` is 3):

```
The number is 5 or less.
```

Ladder If-Else Statement:- Also known as an "else-if" ladder, it is used when multiple conditions need to be checked in sequence. If one of the conditions is true, the corresponding block executes, and the rest are skipped.

```
int score = 85;
if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else if (score >= 70) {
    System.out.println("Grade: C");
} else if (score >= 60) {
    System.out.println("Grade: D");
} else {
    System.out.println("Grade: F");
}
```

Output (if `score` is 85):

```
Grade: B
```

Nested If-Else Statement:- In a nested `if-else` statement, an `if` or `else` block contains another `if-else` statement. It's often used for more complex conditions.

```
int age = 25;
int height = 170;

if (age > 18) {
    if (height > 160) {
        System.out.println("Eligible for the competition.");
    } else {
        System.out.println("Not tall enough for the competition.");
    }
} else {
    System.out.println("Too young for the competition.");
}
```

Output (if `age` is 25 and `height` is 170):

```
Eligible for the competition.
```

Switch Statement:- A `switch` statement allows you to select one of many code blocks to execute, based on the value of a variable. It’s often used as an alternative to the if-else ladder for checking multiple conditions based on a single variable.

```
int day = 3;
switch (day) {
    case 1:
        System.out.println("Monday");
        break;
    case 2:
        System.out.println("Tuesday");
        break;
    case 3:
        System.out.println("Wednesday");
        break;
    case 4:
        System.out.println("Thursday");
        break;
    case 5:
        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
        break;
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("Invalid day");
        break;
}
```

Output (if `day` is 3):

```
Wednesday
```

Object Oriented Programming

Class:- A **class** is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that the objects created from it will have. In Java, a class is defined using the `class` keyword.

Object :- An **object** is an instance of a class. It represents a specific example of the class with real values for the attributes. Each object has its own identity, state (values for the attributes), and behavior (methods it can perform).

Static Keyword

Static Variables:- A **static variable** (also known as a class variable) is a variable that belongs to the class, not to instances of the class. There is only one copy of a static variable per class, shared among all instances.

Static Methods:- A **static method** is a method that belongs to the class rather than to instances of the class. Static methods can be called without creating an instance of the class. However, they can only access static variables and other static methods directly, as they do not have access to instance variables.

Inheritance

types of Inheritance:- Java supports several types of inheritance, although it doesn't support **multiple inheritance** (inheriting from more than one superclass) directly due to ambiguity issues. The types of inheritance are:

- **Single Inheritance**: A subclass inherits from only one superclass.

```
class A { }
class B extends A { }  // Single inheritance
```

- **Multilevel Inheritance**: A class is derived from a superclass, which is itself derived from another superclass.

```
class A { }
class B extends A { }
class C extends B { }  // Multilevel inheritance
```

- **Hierarchical Inheritance**: Multiple classes inherit from the same superclass.

```
class A { }
class B extends A { }
class C extends A { }  // Hierarchical inheritance
```

Java does **not support multiple inheritance** (where a subclass inherits from more than one superclass) to avoid **ambiguity** or the “diamond problem.” However, multiple inheritance can be achieved through **interfaces**.

Constructor Chaining:- Constructor chaining is a process of calling one constructor from another constructor within the same class or from a superclass. It helps in setting up the object’s initial state through inheritance. In Java, constructor chaining can be done in two ways:

- **Within the Same Class**: Using the `this()` keyword to call another constructor in the same class.
- **From a Superclass**: Using the `super()` keyword to call a superclass constructor.

Polymorphism

Method Overloading:- Method overloading is a form of compile-time polymorphism, where multiple methods have the same name but different parameters within the same class. The compiler differentiates them by their parameter lists, which include different numbers, types, or orders of parameters.

Method overriding:- Method overriding is a form of runtime polymorphism, where a subclass provides a specific implementation for a method that is already defined in its superclass. The method in the subclass has the same name, return type, and parameters as the method in the superclass, but provides a new implementation.

Dynamic Polymorphism:- Dynamic polymorphism is achieved through **method overriding** and **late binding**. In dynamic polymorphism, the method to be executed is determined at runtime rather than compile-time. This is often achieved through polymorphic references—using a superclass reference to refer to a subclass object.

Encapsulation

Getter& Setter:- Getters and **setters** are methods used to access and update the private fields (variables) of a class. This practice encapsulates the data, allowing controlled access to class properties while keeping the actual variables private. This control helps maintain data integrity and security by adding logic in these methods, such as validation.

Final Keyword:- The `final` keyword in Java is used to create constants and restrict certain operations. It can be applied to variables, methods, and classes, each with specific effects.

Collection Framework

List Interface:- The `List` interface represents an **ordered collection** (also called a sequence) that allows **duplicate elements**. A `List` maintains the order in which elements are inserted, and you can access the elements via an index. Some commonly used implementations of the `List` interface are `ArrayList`, `LinkedList`, and `Vector`.

- **Type of List**:-ArrayList,LinkedList,Vector(Legacy class)
- **Key Features of List**
    - Allows duplicates.
    - Maintains insertion order.
    - Elements can be accessed by their index.
    - Supports positional access and insertion.

Set Interface:- The `Set` interface represents a **collection** that does not allow **duplicate elements**. It is an unordered collection, meaning there is no guarantee of the order in which elements are stored. `Set` is often used when the uniqueness of the elements is more important than their order.

- **Key Features of Set**:
    - Does not allow duplicates.
    - Does not maintain any order (in most cases).
    - Elements are stored uniquely.

Map Interface:- The `Map` interface represents a collection of **key-value pairs**. A `Map` does not allow duplicate keys, but it can have duplicate values. It allows you to map a unique key to a specific value. Common implementations of the `Map` interface include **`HashMap`**, **`LinkedHashMap`**, and **`TreeMap`**.

Exception handling

Purpose of Exception Handling:- Exception handling is a mechanism in Java (and other programming languages) that deals with runtime errors, enabling the program to continue its execution smoothly. The main purposes of exception handling are:

Try-Catch block:- The `try-catch` block is used to handle exceptions in Java. Code that might throw an exception is placed inside the `try` block, and the exception handling code is written inside the `catch` block.

Finally keyword:- The `finally` block is used to execute important code (like resource cleanup) whether or not an exception occurs. It is optional and is always executed after the `try` and `catch` blocks, even if an exception is thrown or not.

Types Of Exception:- Exceptions in Java are categorized into two main types:

- **Checked Exceptions**:
    - Checked exceptions are exceptions that are checked at compile time.
    - These exceptions are subclasses of `Exception` (except `RuntimeException` and its subclasses).

    - The programmer is required to handle checked exceptions using a `try-catch` block or declare them using the `throws` keyword.
    - Example: `IOException`, `SQLException`, `ClassNotFoundException`.
- **Unchecked Exceptions**:
    - Unchecked exceptions are exceptions that are not checked at compile time.
    - These exceptions are subclasses of `RuntimeException`.
    - These exceptions do not need to be explicitly handled or declared.
    - Example: `ArithmeticException`, `NullPointerException`, `ArrayIndexOutOfBoundsException`.

Throw Keyword:- The `throw` keyword is used to explicitly throw an exception from a method or block of code. It is used when a certain condition is met, and you want to intentionally throw an exception, either predefined or custom.

Throws Keyword:- The `throws` keyword is used in a method signature to indicate that the method may throw one or more exceptions. It is used for **checked exceptions** that are not handled within the method but are passed on to the caller.

Abstraction

Abstract class:- An **abstract class** is a class that cannot be instantiated directly. It is designed to be inherited by other classes, providing a base for them to build upon. An abstract class can contain both **abstract methods** (methods without a body) and **concrete methods** (methods with a body). It allows you to define common behavior while leaving specific details for subclasses.

Interface:- An **interface** is a reference type in Java, similar to a class, that can contain only **abstract methods** (methods without a body) and **constant fields** (i.e., static final variables). Interfaces are used to define a contract that other classes can implement. Unlike abstract classes, interfaces do not provide any method implementation (unless using default methods, introduced in Java 8).

WHAT IS DEPENDENCY ?

In Spring Boot (and Java in general), a *dependency* is an external library or module that your application needs to function properly. These dependencies are included in the project and automatically managed, making it easier to reuse code and avoid reinventing the wheel for common functionalities (e.g., JSON processing, database interaction, security, etc.).

```
+---------------------------------------------+
|                Project                 |
|              (pom.xml)                  |
+---------------------------------------------+
            |        |         |
            |        |         |
            v        v         v
+---------------+  +---------------+  +---------------+
|  Dependency  |  |  Dependency  |  |  Dependency  |
|            |  |            |  |            |
|  <groupId>   |  |  <groupId>   |  |  <groupId>   |
|  <artifactId> |  |  <artifactId> |  |  <artifactId> |
|  <version>   |  |  <version>   |  |  <version>   |
+---------------+  +---------------+  +---------------+
            |        |         |
            v        v         v
+-------------------------------------------------------+
|                Maven Repository                |
|              (Local / Central Repo)             |
+-------------------------------------------------------+
            |
            v
+-------------------------------------------------------+
|              Dependency Download                  |
|        (Added to Project Build Path)              |
+-------------------------------------------------------+
```

Using Maven

In Maven, dependencies are managed in the `pom.xml` file. Each dependency is defined with its group ID, artifact ID, and version, like this:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

To import a dependency:

1. Open your `pom.xml` file.
2. Inside the `<dependencies>` section, add the `<dependency>` tags for any libraries you need.
3. Save the file. Maven will download the dependencies and include them in your project.

WHAT IS GROUPID & ARTIFACTID?

Group ID (`groupId`)

The `groupId` represents the *group*, *organization*, or *company* that the project belongs to. It is often structured as a reversed domain name (e.g., `com.example`, `org.springframework.boot`) to prevent naming conflicts.

For example:

- `org.springframework.boot`: Group ID for Spring Boot libraries.
- `com.fasterxml.jackson.core`: Group ID for Jackson libraries, which are commonly used for JSON processing.

2. Artifact ID (`artifactId`)

The `artifactId` is the *name of the project* or *library*. This ID should be unique within the `groupId` namespace, as it represents the specific component you're including.

For example:

- `spring-boot-starter-web`: Artifact ID for the Spring Boot web starter (used to create web applications).
- `spring-boot-starter-data-jpa`: Artifact ID for Spring Data JPA, used for working with databases in Spring.

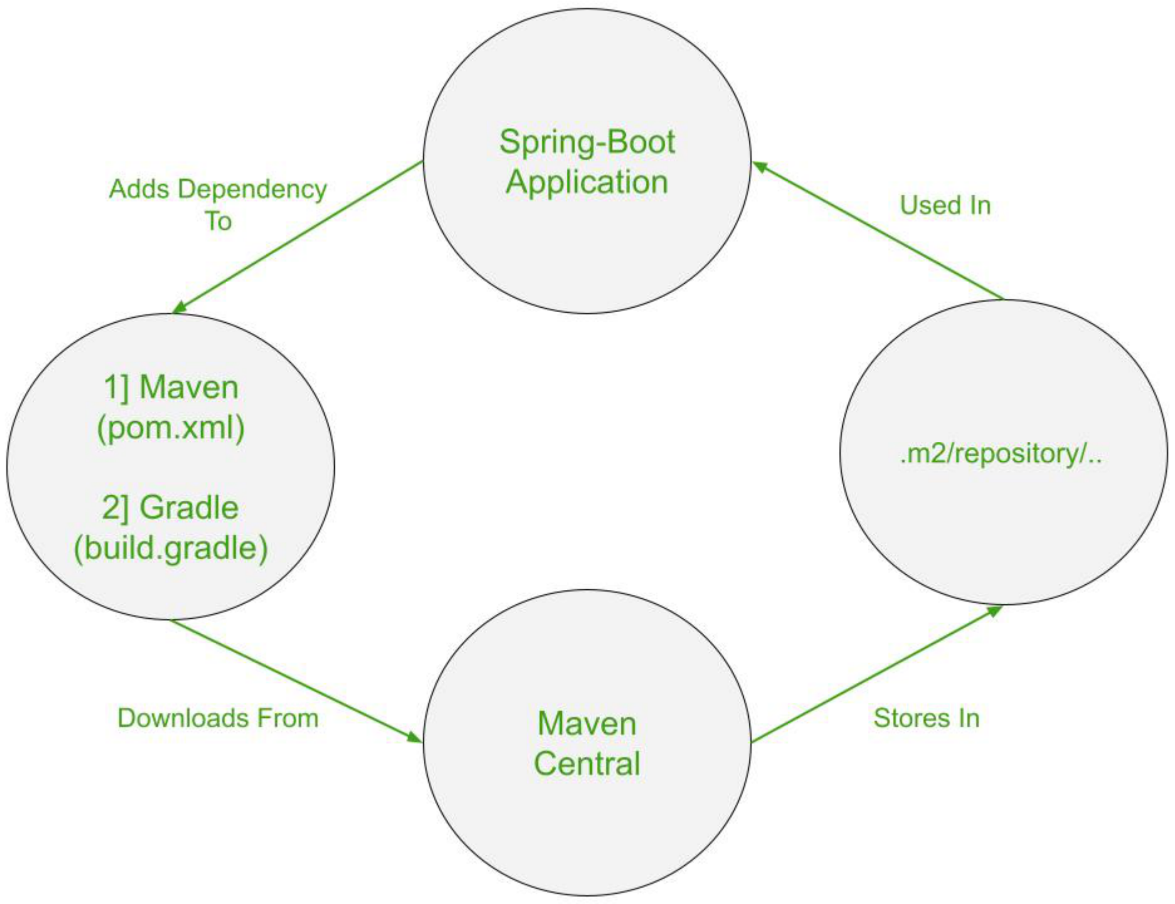
Common Dependencies in Spring Boot

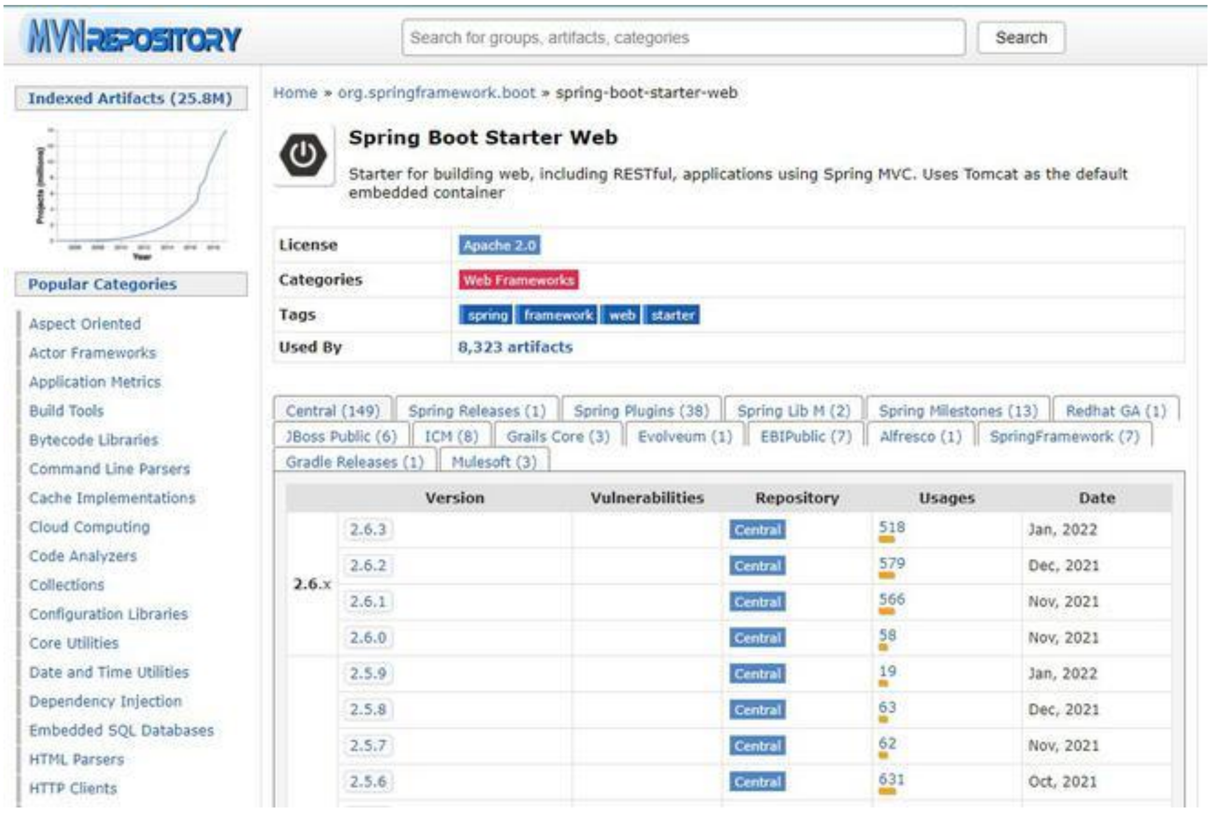
Some common dependencies for Spring Boot applications include:

- **spring-boot-starter-web**: For building web applications, including RESTful services.
- **spring-boot-starter-data-jpa**: For working with relational databases via JPA.
- **spring-boot-starter-security**: For adding authentication and authorization to your application.

Managing Dependencies

To manage dependencies in your Spring Boot application, you can either apply the io.spring.dependency-management plugin or use Maven's pom support.




MVNREPOSITORY
Search for groups, artifacts, categories
Search
Indexed Artifacts (25.8M)
Projects (millions)
Year

Popular Categories

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries
- Core Utilities
- Date and Time Utilities
- Dependency Injection
- Embedded SQL Databases
- HTML Parsers
- HTTP Clients

Home » org.springframework.boot » spring-boot-starter-web

Spring Boot Starter Web

Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container

License	Apache 2.0
Categories	Web Frameworks
Tags	spring framework web starter
Used By	8,323 artifacts

Central (149) | Spring Releases (1) | Spring Plugins (38) | Spring Lib M (2) | Spring Milestones (13) | Redhat GA (1) | JBoss Public (6) | ICM (8) | Grails Core (3) | Evolveum (1) | EBIPublic (7) | Alfresco (1) | SpringFramework (7) | Gradle Releases (1) | Mulesoft (3)

	Version	Vulnerabilities	Repository	Usages	Date
2.6.x	2.6.3		Central	518	Jan, 2022
	2.6.2		Central	579	Dec, 2021
	2.6.1		Central	566	Nov, 2021
	2.6.0		Central	58	Nov, 2021
	2.5.9		Central	19	Jan, 2022
	2.5.8		Central	63	Dec, 2021
	2.5.7		Central	62	Nov, 2021
	2.5.6		Central	631	Oct, 2021

_By Scope

Dependency scope defines when and how a dependency is available in your project. In Maven, there are several scopes:

- `compile` (default):
    - The dependency is available at compile time and runtime, and it is included in the final packaged application.
    - Used for most application dependencies.
    - Example: `spring-boot-starter-web`.
- `provided`:
    - The dependency is required for compilation but is not included in the final package (JAR/WAR), as it is expected to be provided by the runtime environment.
    - Commonly used for Java EE or Servlet API libraries, where the server provides these dependencies.
    - Example: `javax.servlet:javax.servlet-api`.
- `runtime`:
    - The dependency is only needed at runtime, not at compile time. It's included in the final package.
    - Useful for libraries like JDBC drivers, where the API might be defined elsewhere.
    - Example: database drivers like `mysql:mysql-connector-java`.
- `test`:
    - Available only in the test phase, and not included in the final package.
    - Used for testing frameworks like JUnit and Mockito.
    - Example: `org.junit.jupiter:junit-jupiter`.
- `system`:

  - Similar to `provided`, but the JAR must be explicitly declared with an absolute path on the system (rarely used and not recommended).
- **`import`** (used in Dependency Management):
  - Only used with `<dependencyManagement>` to import dependencies from another BOM (Bill of Materials).
  - Typically used in Spring Boot projects for managing versions of Spring dependencies.

2. By Functionality in Spring Boot

In Spring Boot, dependencies are often grouped by their functionality:

- **Starter Dependencies**:
  - Spring Boot offers several "starter" dependencies that bundle related libraries for specific functionalities, making it easy to add multiple dependencies at once.
  - Examples:
    - **`spring-boot-starter-web`**: For building web applications with Spring MVC, Tomcat, Jackson, etc.
    - **`spring-boot-starter-data-jpa`**: For JPA-based data access with Hibernate.
    - **`spring-boot-starter-security`**: For implementing security features.
- **Core Dependencies**:
  - Fundamental Spring dependencies, such as `spring-core`, `spring-context`, and `spring-beans`, which are essential for the Spring Framework itself.
- **Third-Party Dependencies**:
  - External libraries that you may need, such as JWT libraries for token management or Apache Commons for utility functions.
  - Example: `io.jsonwebtoken:jjwt`.

3. By Dependency Relationship

Dependencies can also be classified based on their relationship to other dependencies in the project:

- **Direct Dependencies**:
  - Dependencies explicitly declared in `pom.xml`.
  - Example: If you add `spring-boot-starter-web` in `pom.xml`, it is a direct dependency.
- **Transitive Dependencies**:
  - Dependencies of a dependency. For example, `spring-boot-starter-web` brings in multiple transitive dependencies like `spring-web`, `spring-webmvc`, `tomcat`, etc.
  - Maven automatically includes transitive dependencies unless explicitly excluded.

Exception Handling

Exception handling in Spring Boot is a mechanism that allows developers to manage and respond to errors and exceptions in a structured and user-friendly way, especially in REST APIs. It provides tools and strategies to catch, handle, and respond to exceptions gracefully, allowing you to define custom messages or HTTP statuses rather than exposing stack traces to the user. This improves the user experience and application security.

Key Components of Exception Handling in Spring Boot

1. **Try-Catch Blocks**
   - Used within methods to handle exceptions locally. For example, if a specific error is expected in a service method, you can use a `try-catch` block to handle it right there.
2. **@ExceptionHandler Annotation**
   - This annotation is used at the method level within a `@Controller` or `@RestController`. It allows you to define specific exception-handling methods for individual controllers.
   - When an exception of a certain type is thrown, the corresponding `@ExceptionHandler` method is triggered.

```
@ExceptionHandler(EmployeeNotFoundException.class)
public ResponseEntity<String>
handleEmployeeNotFound(EmployeeNotFoundException ex) {
    return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
}
```

3. **@ControllerAdvice**
   - `@ControllerAdvice` is a more centralized approach to handle exceptions across all controllers.
   - It acts as a global exception handler, capturing exceptions thrown by any controller in the application.
   - Using `@ExceptionHandler` methods within a `@ControllerAdvice` class allows you to manage exceptions from a single place.

```
@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EmployeeNotFoundException.class)
    public ResponseEntity<String>
handleEmployeeNotFound(EmployeeNotFoundException ex) {
        return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleGenericException(Exception ex)
{
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body("An error occurred: " +
ex.getMessage());
    }
}
```

4.
5. **ResponseEntityExceptionHandler**
   - This is a built-in Spring class that provides default handling for standard Spring MVC exceptions (like `HttpRequestMethodNotSupportedException`, `HttpMediaTypeNotSupportedException`, etc.).
   - You can extend this class to customize exception responses for specific Spring MVC errors.
6. **@ResponseStatus Annotation**
   - This annotation allows you to specify the HTTP status code for a specific exception. It can be applied directly to custom exception classes, so when the exception is thrown, the specified status code is returned.

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class EmployeeNotFoundException extends RuntimeException {
    public EmployeeNotFoundException(String message) {
        super(message);
    }
}
```

7.

Example: Exception Handling in a Spring Boot REST API

Let's say you have an endpoint that retrieves an employee by ID. If the ID is invalid, you want to throw a custom `EmployeeNotFoundException` and handle it gracefully.

Define the Custom Exception

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class EmployeeNotFoundException extends RuntimeException {
    public EmployeeNotFoundException(String message) {
        super(message);
    }
}
```

Use the Exception in the Controller or Service

```
@GetMapping("/employees/{id}")
public ResponseEntity<Employee> getEmployeeById(@PathVariable Long id)
{
    Employee employee = employeeService.findById(id)
                       .orElseThrow(() -> new
EmployeeNotFoundException("Employee not found with ID: " + id));
    return ResponseEntity.ok(employee);
}
```

1.

Handle the Exception in @ControllerAdvice

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(EmployeeNotFoundException.class)
    public ResponseEntity<String>
handleEmployeeNotFound(EmployeeNotFoundException ex) {
        return
ResponseEntity.status(HttpStatus.NOT_FOUND).body(ex.getMessage());
    }
}
```

2. This setup allows you to:

- Return a `404 Not Found` HTTP status when an employee with the specified ID is not found.
- Show a custom error message to the client instead of a generic server error.

Benefits of Exception Handling in Spring Boot

- **Centralized error handling** for better code organization.
- **User-friendly error responses** by avoiding raw stack traces and providing custom messages.
- **Customizable HTTP status codes** that align with REST API conventions (like 404 for "not found," 400 for "bad request," etc.).
- **Enhanced security** by preventing sensitive information from being exposed in error messages.

With Spring Boot's exception handling features, you can create a more stable, user-friendly, and resilient application.

```
@GetMapping("/get/{id}")
        public ResponseEntity<?> getEmployee(@PathVariable Long id){
                try {
          Employee employee = service.getEmployee(id);
          return ResponseEntity.ok(employee);
      } catch (ResourceNotFoundException e) {
          return
ResponseEntity.status(HttpStatus.NOT_FOUND).body("Resource not
found");
      } catch (EmployeeNotFoundException e) {
                        // TODO Auto-generated catch block
         return
ResponseEntity.status(HttpStatus.NOT_FOUND).body("employee not
found");

                }
        }
```

```
public Employee getEmployee(Long id) throws
EmployeeNotFoundException {
                // TODO Auto-generated method stub
                 if (id <= 0) {
                  throw new EmployeeNotFoundException("Employee not
found with ID: " + id);
               }
                Employee byId = repository.findById(id).orElseThrow(()->new
ResourceNotFoundException("Employee not found with user id "+id));


                return byId;
        }
```

```
package com.example.exception;
public class EmployeeNotFoundException extends Exception {
        public EmployeeNotFoundException(String message) {
      super(message);
   }
}
```

```
package com.example.exception;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
@RestControllerAdvice
public class GlobalExceptionHandler {
         @ExceptionHandler(ResourceNotFoundException.class)
            public ResponseEntity<?>
handlerResourceNotFoundException(ResourceNotFoundException ex){
                String message = ex.getMessage();
                return new
ResponseEntity<>(message,HttpStatus.NOT_FOUND);
```

```
            }
         @ExceptionHandler(EmployeeNotFoundException.class)
            public ResponseEntity<String>
handleUserNotFoundException(EmployeeNotFoundException ex) {
                  String message = ex.getMessage();
       return new ResponseEntity<>(message,HttpStatus.NOT_FOUND);
         }
}
```

JWT

- **What is Jwt?**

->JWT, or JSON Web Token, is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued.

How JWT Works

JWTs differ from other web tokens in that they contain a set of claims. Claims are used to transmit information between two parties. What these claims are depends on the use case at hand. For example, a claim may assert who issued the token, how long it is valid for, or what permissions the client has been granted.

A JWT is a string made up of three parts, separated by dots (.), and serialized using base64. In the most common serialization format, compact serialization, the JWT looks something like this:

xxxxx.yyyyy.zzzzz.

Once decoded, you will get two JSON strings:

1. The **header** and the **payload.**
2. The **signature.**

The **JOSE (JSON Object Signing and Encryption) header** contains the type of token — JWT in this case — and the signing algorithm.

The **payload** contains the claims. This is displayed as a JSON string, usually containing no more than a dozen fields to keep the JWT compact. This information is typically used by the server to verify that the user has permission to perform the action they are requesting.

There are no mandatory claims for a JWT, but overlaying standards may make claims mandatory. For example, when using JWT as bearer access token under OAuth2.0, iss, sub, aud, and exp must be present. some are more common than others.

The **signature** ensures that the token hasn't been altered. The party that creates the JWT signs the header and payload with a secret that is known to both the issuer and receiver, or with a private key known only to the sender. When the token is used, the receiving party verifies that the header and payload match the signature.

Class for provide username and password

```
package com.example.security;



public class AuthenticationRequest {

    private String username;
    private String password;
     public String getUsername() {
          return username;
     }
     public void setUsername(String username) {
          this.username = username;
     }
     public String getPassword() {
          return password;
     }
     public void setPassword(String password) {
          this.password = password;
     }


}
```

```
==========================================================


package com.example.security;


import com.auth0.jwt.JWT;
import com.auth0.jwt.algorithms.Algorithm;
import com.auth0.jwt.interfaces.DecodedJWT;
import org.springframework.stereotype.Component;

import java.util.Date;

@Component
public class JwtTokenUtil {

    private static final String SECRET_KEY =
"mySecretKey";
    private static final long EXPIRATION_TIME = 1000 * 60
* 60; // 1 hour

    public String generateToken(String username) {
        Algorithm algorithm =
Algorithm.HMAC256(SECRET_KEY);
        return JWT.create()
                .withSubject(username)
                .withIssuedAt(new Date())
                .withExpiresAt(new
Date(System.currentTimeMillis() + EXPIRATION_TIME))
                .sign(algorithm);
    }

    public String extractUsername(String token) {
        return extractClaims(token).getSubject();
    }

    public DecodedJWT extractClaims(String token) {
        Algorithm algorithm =
Algorithm.HMAC256(SECRET_KEY);
        return JWT.require(algorithm)
                .build()
                .verify(token);
    }


 public boolean isTokenExpired(String token) {
        return 
extractClaims(token).getExpiresAt().before(new Date());
    }

    public boolean validateToken(String token, String 
username) {
        return (username.equals(extractUsername(token)) && 
!isTokenExpired(token));
    }
}
```

```
===========================================================
package com.example.security;


import java.io.IOException;
import java.util.ArrayList;

import
org.springframework.security.core.context.SecurityContextH
older;
import
org.springframework.web.filter.OncePerRequestFilter;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class JwtRequestFilter extends OncePerRequestFilter
{

    private final JwtTokenUtil jwtTokenUtil;

    public JwtRequestFilter(JwtTokenUtil jwtTokenUtil) {
        this.jwtTokenUtil = jwtTokenUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest
request,HttpServletResponse response, FilterChain
filterChain) throws ServletException, IOException {
        String jwtToken =
request.getHeader("Authorization");

        if (jwtToken != null &&
jwtToken.startsWith("Bearer ")) {
            // Extract the token from the Authorization
header
            String token = jwtToken.substring(7);
            String username =
jwtTokenUtil.extractUsername(token);

            if (username != null &&
SecurityContextHolder.getContext().getAuthentication() ==
null) {
                // Validate token and set authentication
if valid
                if (jwtTokenUtil.validateToken(token,
username)) {

SecurityContextHolder.getContext().setAuthentication(new
org.springframework.security.authentication.UsernamePasswo
rdAuthenticationToken(username, null, new ArrayList<>()));
                }
            }
        }
        filterChain.doFilter(request, response);
    }
}
```

```
==========================================================
package com.example.security;


import org.springframework.context.annotation.Bean;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.web.builder
s.HttpSecurity;
import org.springframework.security.core.userdetails.User;
import
org.springframework.security.core.userdetails.UserDetailsS
ervice;
import
org.springframework.security.crypto.bcrypt.BCryptPasswordE
ncoder;
import
org.springframework.security.crypto.password.PasswordEncod
er;
import
org.springframework.security.provisioning.InMemoryUserDeta
ilsManager;
import
org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePa
sswordAuthenticationFilter;
import
org.springframework.security.config.annotation.web.configu
ration.EnableWebSecurity;

@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

    private final JwtTokenUtil jwtTokenUtil;

    // Constructor-based injection of JwtTokenUtil
    public WebSecurityConfig(JwtTokenUtil jwtTokenUtil) {
        this.jwtTokenUtil = jwtTokenUtil;
    }

    @Bean
    public SecurityFilterChain
securityFilterChain(HttpSecurity http) throws Exception {
        // Disable CSRF for simplicity in this example
        http.csrf().disable()
            .authorizeRequests()
                .requestMatchers("/authenticate",
"/register","create","/get/**", "/delete/**").permitAll()
// Allow public access to login/register endpoints
                .anyRequest().authenticated()  // Require
authentication for all other requests
            .and()
            .addFilterBefore(new
JwtRequestFilter(jwtTokenUtil),
UsernamePasswordAuthenticationFilter.class);  // Register
JwtRequestFilter before
UsernamePasswordAuthenticationFilter

        return http.build();
    }

    @Bean
    public UserDetailsService userDetailsService() {
        // Example in-memory user (replace with a real
user service in production)
        return new InMemoryUserDetailsManager(
            User.withUsername("root")

.password(passwordEncoder().encode("admin"))
                .roles("USER")
                .build()
        );
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

SQL

Content:-

max

count

having

inner join

left join

join

union

case

Employee Table

id	name	salary	department	branch
1	Alice	60000	HR	1
2	Bob	40000	IT	2
3	Charlie	30000	IT	2
4	Diana	70000	HR	1
5	Edward	20000	Sales	NULL

Branch Table

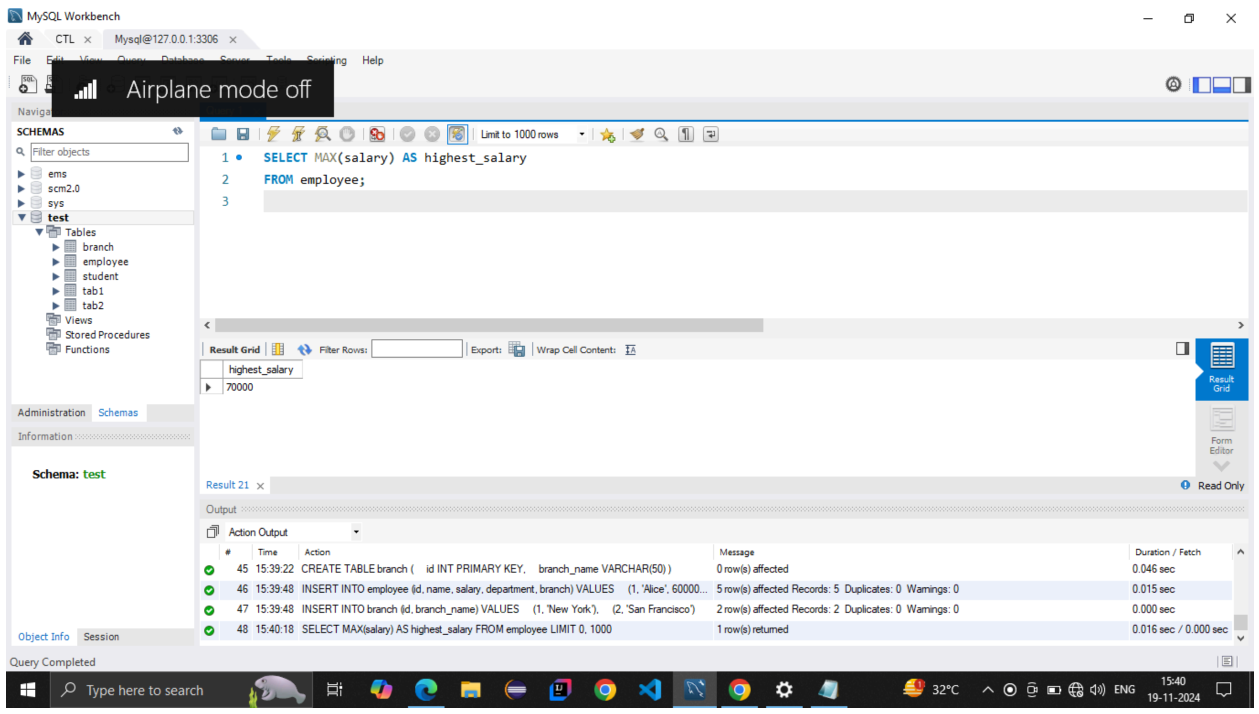
ID	Branch_Name
1	New York
2	San Francisco

1. MAX

Definition: The `MAX` function returns the highest value in a column.

Query: Find the employee with the highest salary.

```sql
SELECT MAX(salary) AS highest_salary
FROM employee;
```



```sql
SELECT MAX(salary) AS highest_salary
FROM employee;
```

```
highest_salary
70000
```

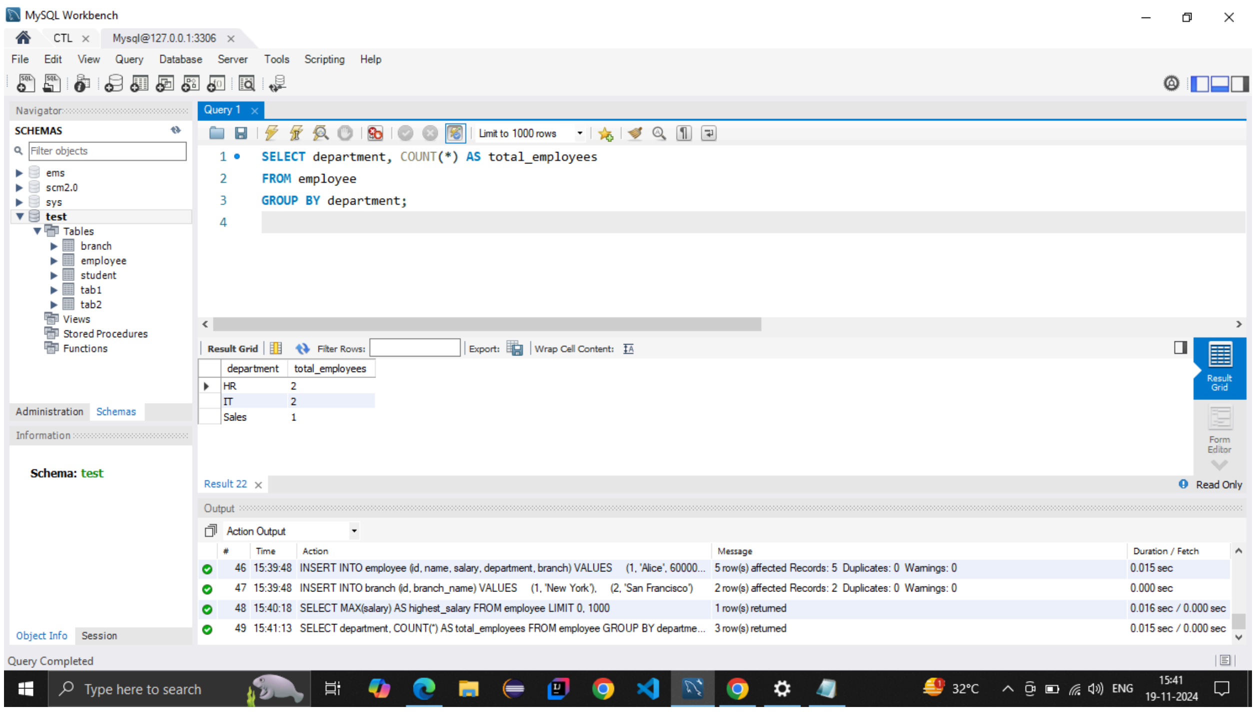
```
#   Time      Action                                                                    Message                                                      Duration / Fetch
45  15:39:22  CREATE TABLE branch (    id INT PRIMARY KEY,    branch_name VARCHAR(50) )  0 row(s) affected                                            0.046 sec
46  15:39:48  INSERT INTO employee (id, name, salary, department, branch) VALUES    (1, 'Alice', 60000...  5 row(s) affected Records: 5  Duplicates: 0  Warnings: 0  0.015 sec
47  15:39:48  INSERT INTO branch (id, branch_name) VALUES    (1, 'New York'),    (2, 'San Francisco')  2 row(s) affected Records: 2  Duplicates: 0  Warnings: 0  0.000 sec
48  15:40:18  SELECT MAX(salary) AS highest_salary FROM employee LIMIT 0, 1000          1 row(s) returned                                            0.016 sec / 0.000 sec
```

2. COUNT

Definition: The COUNT function calculates the number of rows that match a condition.

Query: Count the total number of employees in each department.

```
SELECT department, COUNT(*) AS total_employees
FROM employee
GROUP BY department;
```



```
SELECT department, COUNT(*) AS total_employees
FROM employee
GROUP BY department;
```

department	total_employees
HR	2
IT	2
Sales	1

```
#   Time      Action                                                                                  Message                                                     Duration / Fetch
46  15:39:48  INSERT INTO employee (id, name, salary, department, branch) VALUES    (1, 'Alice', 60000...  5 row(s) affected Records: 5  Duplicates: 0  Warnings: 0   0.015 sec
47  15:39:48  INSERT INTO branch (id, branch_name) VALUES    (1, 'New York'),    (2, 'San Francisco')     2 row(s) affected Records: 2  Duplicates: 0  Warnings: 0   0.000 sec
48  15:40:18  SELECT MAX(salary) AS highest_salary FROM employee LIMIT 0, 1000                            1 row(s) returned                                           0.016 sec / 0.000 sec
49  15:41:13  SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...           3 row(s) returned                                           0.015 sec / 0.000 sec
```

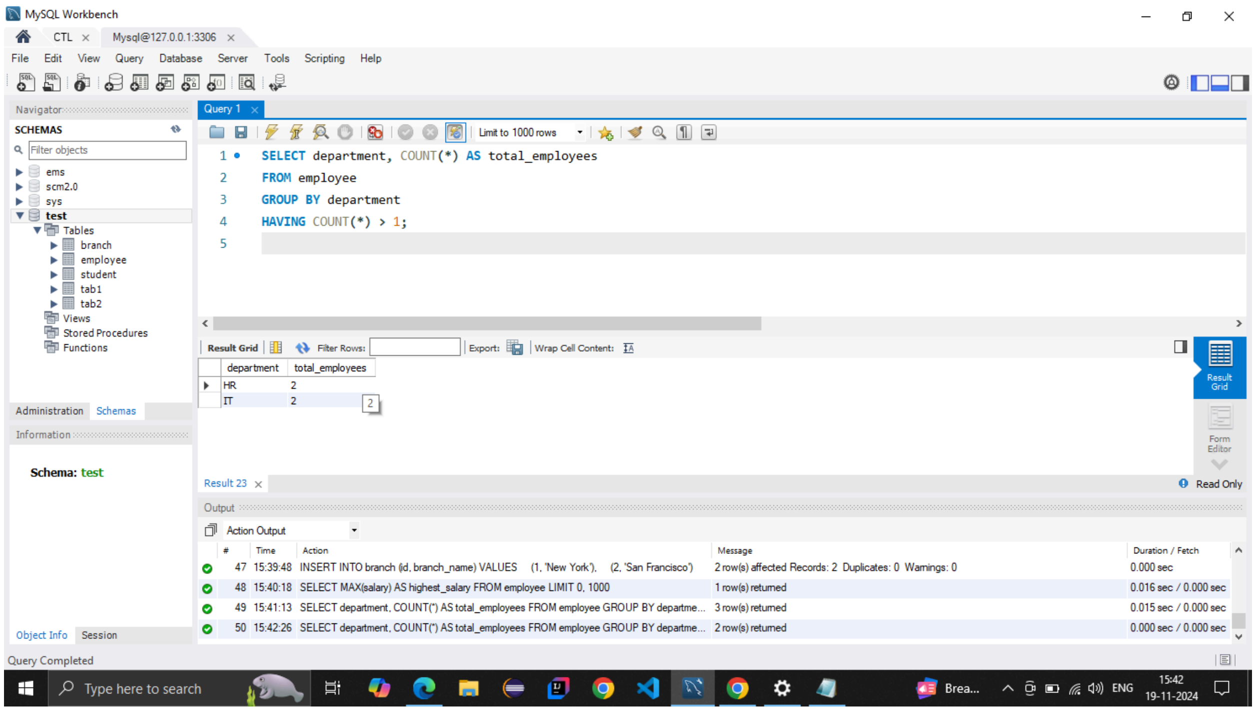
3. HAVING

Definition: The HAVING clause filters groups after aggregation (unlike WHERE, which filters rows).

Query: List departments with more than 5 employees.

```
SELECT department, COUNT(*) AS total_employees
```

```sql
FROM employee
GROUP BY department
HAVING COUNT(*) > 5;
```



```sql
SELECT department, COUNT(*) AS total_employees
FROM employee
GROUP BY department
HAVING COUNT(*) > 1;
```

department	total_employees
HR	2
IT	2

#	Time	Action	Message	Duration / Fetch
47	15:39:48	INSERT INTO branch (id, branch_name) VALUES (1, 'New York'), (2, 'San Francisco')	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0	0.000 sec
48	15:40:18	SELECT MAX(salary) AS highest_salary FROM employee LIMIT 0, 1000	1 row(s) returned	0.016 sec / 0.000 sec
49	15:41:13	SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...	3 row(s) returned	0.015 sec / 0.000 sec
50	15:42:26	SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...	2 row(s) returned	0.000 sec / 0.000 sec

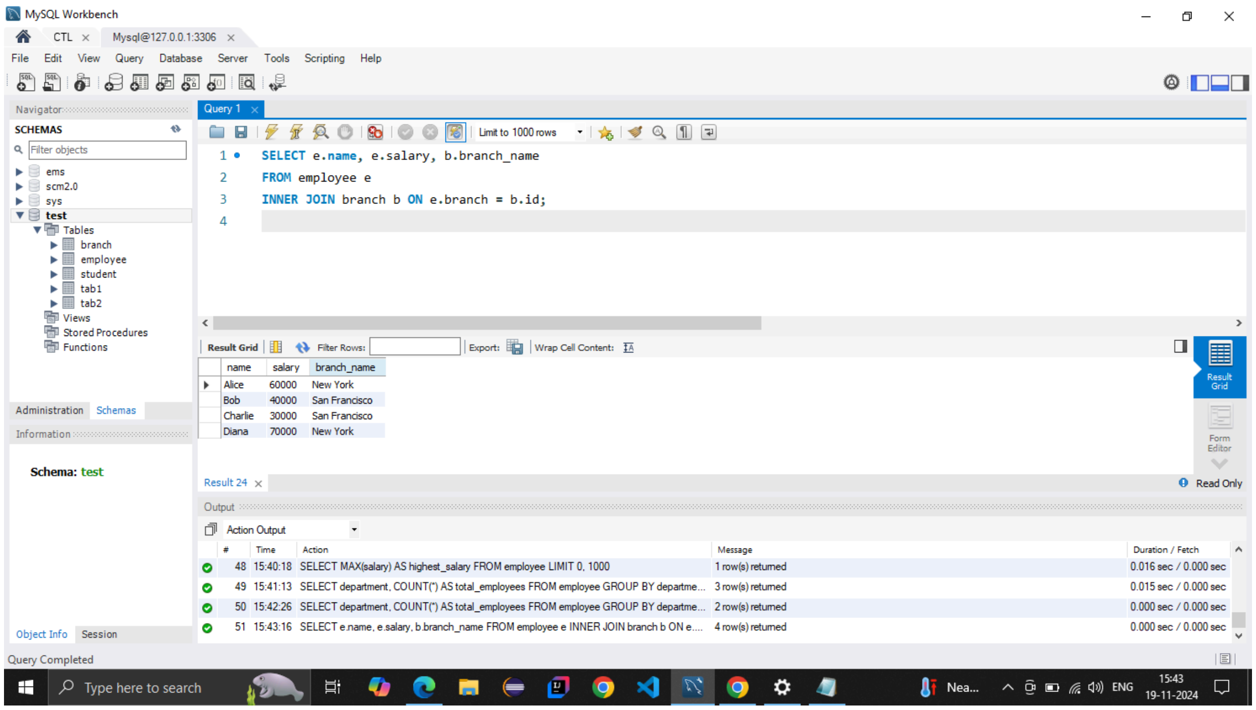
Query Completed

4. INNER JOIN

Definition: Combines rows from two tables where there is a match in the specified columns.

Query: Get employees' salaries along with their branch information from a `branch` table.

```sql
SELECT e.name, e.salary, b.branch_name
FROM employee e
INNER JOIN branch b ON e.branch = b.id;
```



```sql
SELECT e.name, e.salary, b.branch_name
FROM employee e
INNER JOIN branch b ON e.branch = b.id;
```

name	salary	branch_name
Alice	60000	New York
Bob	40000	San Francisco
Charlie	30000	San Francisco
Diana	70000	New York

```
#   Time      Action                                                                      Message            Duration / Fetch
48  15:40:18  SELECT MAX(salary) AS highest_salary FROM employee LIMIT 0, 1000              1 row(s) returned  0.016 sec / 0.000 sec
49  15:41:13  SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...  3 row(s) returned  0.015 sec / 0.000 sec
50  15:42:26  SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...  2 row(s) returned  0.000 sec / 0.000 sec
51  15:43:16  SELECT e.name, e.salary, b.branch_name FROM employee e INNER JOIN branch b ON e....  4 row(s) returned  0.000 sec / 0.000 sec
```

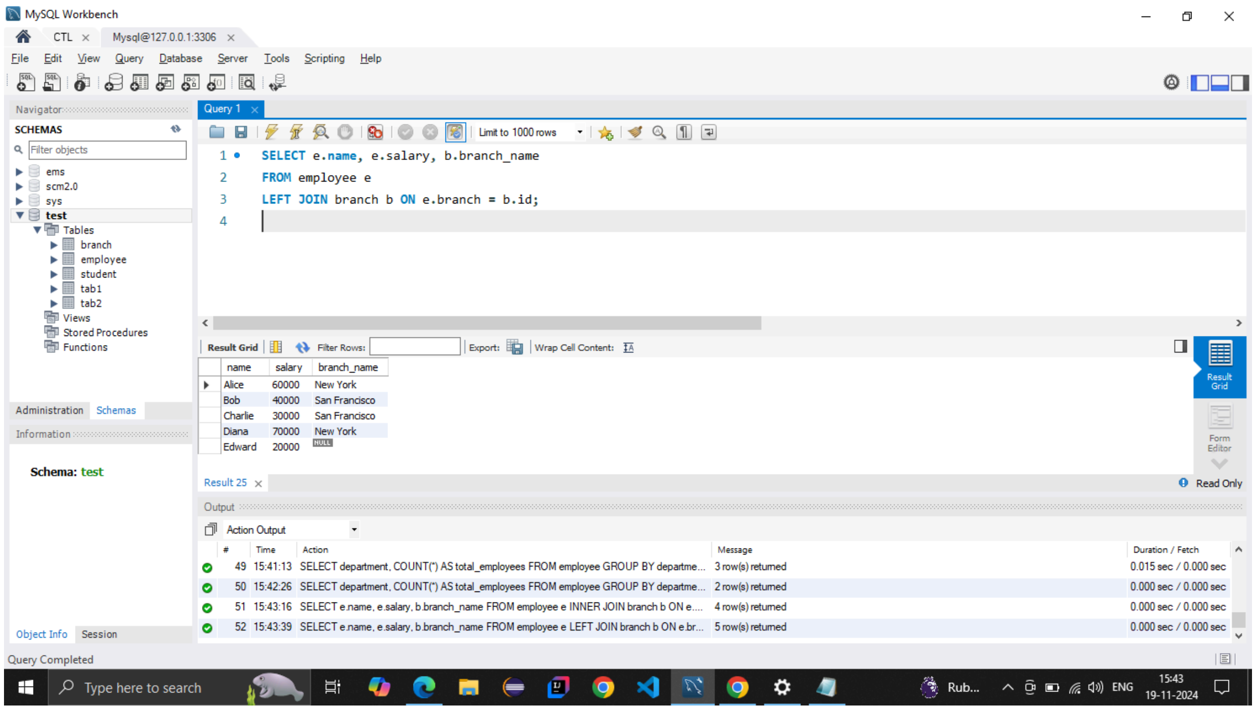
Query Completed

5. LEFT JOIN

Definition: Returns all rows from the left table and the matching rows from the right table. If no match, NULL is returned for the right table's columns.

Query: Get all employees and their branch information, including employees not assigned to any branch.

```sql
SELECT e.name, e.salary, b.branch_name
FROM employee e
LEFT JOIN branch b ON e.branch = b.id;
```



```
SELECT e.name, e.salary, b.branch_name
FROM employee e
LEFT JOIN branch b ON e.branch = b.id;
```

name	salary	branch_name
Alice	60000	New York
Bob	40000	San Francisco
Charlie	30000	San Francisco
Diana	70000	New York
Edward	20000	NULL

```
#   Time      Action                                                                          Message             Duration / Fetch
49  15:41:13  SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...  3 row(s) returned   0.015 sec / 0.000 sec
50  15:42:26  SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...  2 row(s) returned   0.000 sec / 0.000 sec
51  15:43:16  SELECT e.name, e.salary, b.branch_name FROM employee e INNER JOIN branch b ON e....  4 row(s) returned   0.000 sec / 0.000 sec
52  15:43:39  SELECT e.name, e.salary, b.branch_name FROM employee e LEFT JOIN branch b ON e.br...  5 row(s) returned   0.000 sec / 0.000 sec
```

6. JOIN

Definition: A general term referring to the combination of rows from two or more tables. SQL supports multiple types of joins like `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, etc.

---

7. UNION

Definition: Combines the results of two or more `SELECT` queries into a single result set. Duplicate rows are removed unless `UNION ALL` is used.

Query: Combine employees earning above 50,000 with those in the "HR" department.

```
SELECT name, salary, department

FROM employee
```

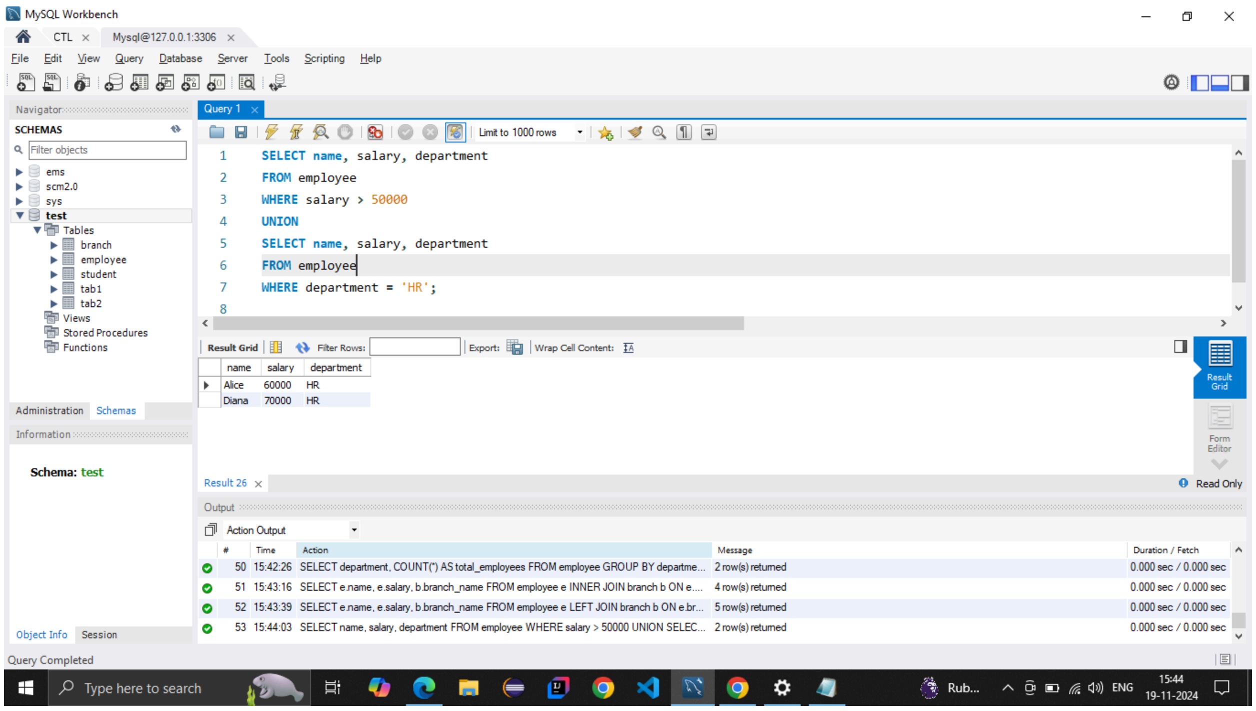
```
WHERE salary > 50000

UNION

SELECT name, salary, department

FROM employee

WHERE department = 'HR';
```



```
SELECT name, salary, department
FROM employee
WHERE salary > 50000
UNION
SELECT name, salary, department
FROM employee
WHERE department = 'HR';
```

name	salary	department
Alice	60000	HR
Diana	70000	HR

```
#   Time      Action                                                                            Message            Duration / Fetch
50  15:42:26  SELECT department, COUNT(*) AS total_employees FROM employee GROUP BY departme...  2 row(s) returned  0.000 sec / 0.000 sec
51  15:43:16  SELECT e.name, e.salary, b.branch_name FROM employee e INNER JOIN branch b ON e....  4 row(s) returned  0.000 sec / 0.000 sec
52  15:43:39  SELECT e.name, e.salary, b.branch_name FROM employee e LEFT JOIN branch b ON e.br...  5 row(s) returned  0.000 sec / 0.000 sec
53  15:44:03  SELECT name, salary, department FROM employee WHERE salary > 50000 UNION SELEC...     2 row(s) returned  0.000 sec / 0.000 sec
```

Query Completed

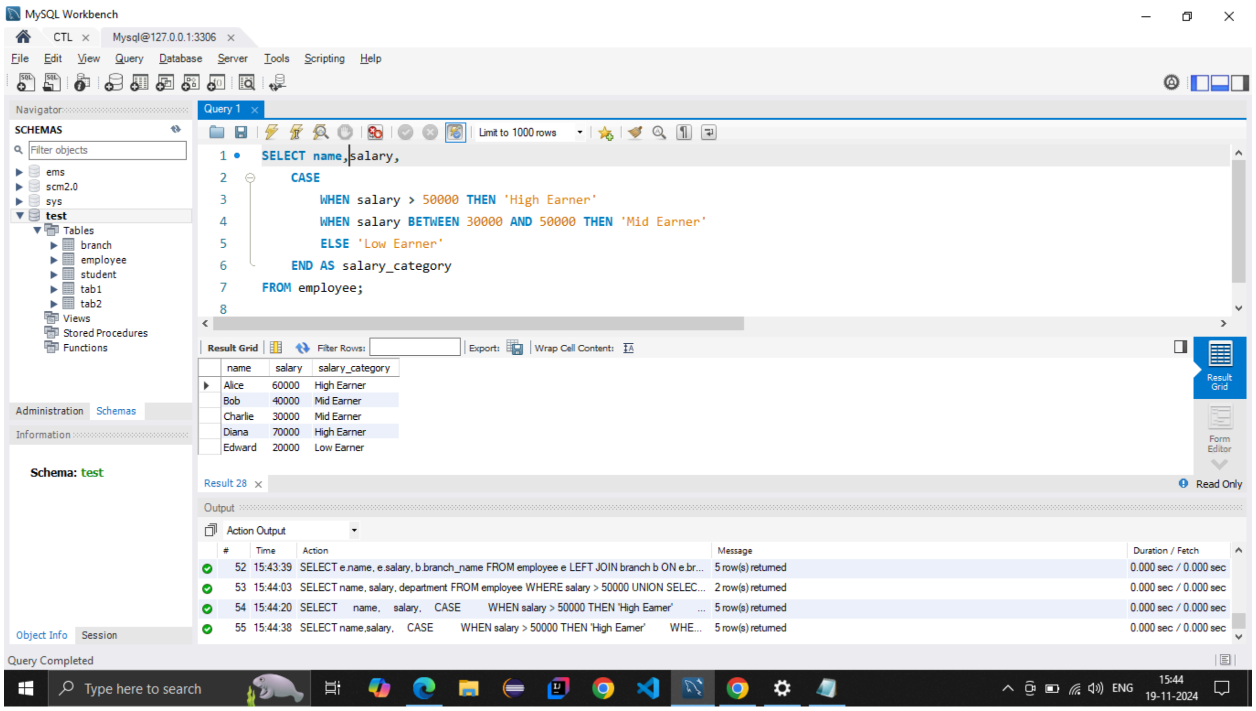
8. CASE

Definition: Provides conditional logic to return different values based on specific conditions.

Query: Categorize employees based on their salary.

```
SELECT
```

```
    name,
    salary,
    CASE
        WHEN salary > 50000 THEN 'High Earner'
        WHEN salary BETWEEN 30000 AND 50000 THEN 'Mid
Earner'
        ELSE 'Low Earner'
    END AS salary_category
FROM employee;
```



```
SELECT name,salary,
    CASE
        WHEN salary > 50000 THEN 'High Earner'
        WHEN salary BETWEEN 30000 AND 50000 THEN 'Mid Earner'
        ELSE 'Low Earner'
    END AS salary_category
FROM employee;
```

name	salary	salary_category
Alice	60000	High Earner
Bob	40000	Mid Earner
Charlie	30000	Mid Earner
Diana	70000	High Earner
Edward	20000	Low Earner

```
#   Time      Action                                                                          Message           Duration / Fetch
52  15:43:39  SELECT e.name, e.salary, b.branch_name FROM employee e LEFT JOIN branch b ON e.br...  5 row(s) returned  0.000 sec / 0.000 sec
53  15:44:03  SELECT name, salary, department FROM employee WHERE salary > 50000 UNION SELEC...    2 row(s) returned  0.000 sec / 0.000 sec
54  15:44:20  SELECT    name,    salary,    CASE       WHEN salary > 50000 THEN 'High Earner'   ...  5 row(s) returned  0.000 sec / 0.000 sec
55  15:44:38  SELECT name,salary,    CASE       WHEN salary > 50000 THEN 'High Earner'      WHE...  5 row(s) returned  0.000 sec / 0.000 sec
```

Query Completed