**Basic Commands:**

* **\c database\_name:** Connects to a specific database.
* **\l:** Lists all databases.
* **\q:** Exits the psql shell.
* **\h command:** Shows help for a specific command.

**Query Commands:**

* **SELECT column\_name FROM table\_name:** Retrieves data from a table.
* **INSERT INTO table\_name (column1, column2) VALUES (value1, value2);:** Inserts data into a table.
* **UPDATE table\_name SET column\_name = new\_value WHERE condition;:** Updates existing data in a table.
* **DELETE FROM table\_name WHERE condition;:** Deletes rows from a table.

**Data Definition Language (DDL) Commands:**

* **CREATE TABLE table\_name (column1 data\_type, column2 data\_type, ...);:** Creates a new table.
* **ALTER TABLE table\_name ADD COLUMN column\_name data\_type;:** Adds a column to a table.
* **ALTER TABLE table\_name DROP COLUMN column\_name;:** Removes a column from a table.
* **DROP TABLE table\_name;:** Deletes a table.

**Data Manipulation Language (DML) Commands:**

* **COMMIT:** Saves changes made in a transaction.
* **ROLLBACK:** Reverts changes made in a transaction.
* **BEGIN;** Starts a new transaction.

**Other Commands:**

* **\d table\_name:** Describes a table's structure.
* **\dt:** Lists all tables.
* **\ds:** Lists all sequences.
* **\df:** Lists all functions.
* **\du:** Lists all users.
* **\x:** Toggles expanded output format.

**Example:**

-- Connect to a database named "mydatabase"

\c mydatabase

-- Create a new table

CREATE TABLE customers (

customer\_id SERIAL PRIMARY KEY,

first\_name TEXT,

last\_name TEXT,

email TEXT

);

-- Insert data into the table

INSERT INTO customers (first\_name, last\_name, email)

VALUES ('John', 'Doe', 'john.doe@example.com');

-- Retrieve data from the table

SELECT \* FROM customers;

**Spring Boot :**

**General Spring Boot Annotations**

1. **@SpringBootApplication**: Combines @Configuration, @EnableAutoConfiguration, and @ComponentScan to bootstrap a Spring Boot application.
2. **@RestController**: Indicates that the class serves RESTful web services and combines @Controller and @ResponseBody.
3. **@RequestMapping**: Maps HTTP requests to handler methods of MVC and REST controllers.
4. **@GetMapping / @PostMapping / @PutMapping / @DeleteMapping**: Specialized versions of @RequestMapping for HTTP GET, POST, PUT, and DELETE requests.
5. **@RequestParam**: Binds a request parameter to a method parameter in a controller.
6. **@PathVariable**: Binds a path variable from the URL to a method parameter.
7. **@RequestBody**: Binds the body of the HTTP request to a method parameter.
8. **@ResponseBody**: Indicates that the return value of a method is serialized directly into the HTTP response body.
9. **@Service**: Marks a class as a Spring service component.
10. **@Repository**: Indicates that the class is a Data Access Object (DAO) and applies exception translation to Spring's data access exceptions.
11. **@Component**: Marks a Java class as a Spring component; used for generic beans.
12. **@Autowired**: Automatically injects dependencies into the class fields or methods.
13. **@Qualifier**: Used alongside @Autowired to specify which bean to inject when multiple implementations are available.
14. **@Configuration**: Marks a class as a source of bean definitions for the Spring container.
15. **@Bean**: Indicates that a method produces a Spring-managed bean to be registered in the application context.
16. **@Conditional**: Specifies conditions for registering a bean based on certain criteria.
17. **@Value**: Injects values from properties or environment variables into class fields.
18. **@PropertySource**: Specifies the location of properties files to load into the Spring Environment.
19. **@EnableAutoConfiguration**: Enables Spring Boot’s auto-configuration feature to configure Spring application based on the dependencies present.
20. **@ComponentScan**: Configures component scanning, allowing Spring to detect and register beans within specified packages.
21. **@EnableScheduling**: Enables Spring’s scheduled task execution capability.
22. **@Scheduled**: Marks a method to be scheduled and executed at fixed intervals.
23. **@EnableAsync**: Enables Spring’s asynchronous method execution.
24. **@Async**: Marks a method to be executed asynchronously.
25. **@Transactional**: Manages transaction boundaries on methods or classes, ensuring consistency and rollback on errors.
26. **@ExceptionHandler**: Defines methods in a controller to handle specific exceptions.
27. **@ControllerAdvice**: Global exception handling for controllers, allowing you to apply @ExceptionHandler methods across all controllers.
28. **@CrossOrigin**: Enables Cross-Origin Resource Sharing (CORS) on controller methods.
29. **@RequestHeader**: Binds a request header to a method parameter in a controller.
30. **@CookieValue**: Binds the value of a cookie to a method parameter.
31. **@RestControllerAdvice**: A specialization of @ControllerAdvice for REST controllers, providing global exception handling.
32. **@Scope**: Defines the scope of a Spring bean (e.g., singleton, prototype).
33. **@Primary**: Specifies that a particular bean should be preferred when multiple beans of the same type are present.
34. **@Profile**: Activates beans only when a particular profile is active.
35. **@Valid**: Triggers validation on method parameters, typically for DTOs.
36. **@EnableJpaRepositories**: Enables JPA repository support in the application.
37. **@Entity**: Marks a class as a JPA entity to be managed by the persistence context.
38. **@Table**: Specifies the table name in the database for a JPA entity.
39. **@Id**: Specifies the primary key field in a JPA entity.
40. **@GeneratedValue**: Defines the strategy for generating primary key values in a JPA entity.
41. **@ManyToOne / @OneToMany / @OneToOne / @ManyToMany**: Defines relationships between JPA entities.

**Spring Data JPA Annotations**

1. **@Entity**: Marks a class as a JPA entity, representing a table in the database.
2. **@Table**: Specifies the table name in the database for a JPA entity. If not specified, the entity name is used as the table name.
3. **@Id**: Specifies the primary key field of the entity.
4. **@GeneratedValue**: Defines the strategy for generating primary key values (e.g., AUTO, IDENTITY, SEQUENCE, TABLE).
5. **@Column**: Maps a field to a column in the database table. Can be used to define column name, type, length, and constraints.
6. **@ManyToOne**: Defines a many-to-one relationship between two entities. The owning side of the relationship.
7. **@OneToMany**: Defines a one-to-many relationship. Typically, the inverse side of a @ManyToOne.
8. **@OneToOne**: Defines a one-to-one relationship between two entities.
9. **@ManyToMany**: Defines a many-to-many relationship between two entities, typically requiring a join table.
10. **@JoinColumn**: Specifies the foreign key column in a relationship. Used to customize the foreign key mapping.
11. **@JoinTable**: Specifies the join table for a many-to-many relationship, allowing customization of the join table's name and columns.
12. **@Embedded**: Indicates that a field is an embedded object that will be part of the entity.
13. **@Embeddable**: Marks a class as embeddable; such classes can be embedded in other entity classes.
14. **@Transient**: Indicates that a field is not to be persisted in the database.
15. **@Fetch**: Specifies how related entities should be fetched (e.g., EAGER or LAZY fetching).
16. **@Version**: Used for optimistic locking; indicates a version field to control concurrent updates.
17. **@Query**: Used to define a custom JPQL or SQL query on a repository method.
18. **@Modifying**: Indicates that a query modifies data (used with @Query for update or delete operations).
19. **@EntityListeners**: Specifies one or more listener classes that should be invoked during entity lifecycle events.
20. **@PrePersist**: Callback method invoked before an entity is persisted.
21. **@PostPersist**: Callback method invoked after an entity is persisted.
22. **@PreUpdate**: Callback method invoked before an entity is updated.
23. **@PostUpdate**: Callback method invoked after an entity is updated.
24. **@PreRemove**: Callback method invoked before an entity is removed.
25. **@PostRemove**: Callback method invoked after an entity is removed.
26. **@PostLoad**: Callback method invoked after an entity is loaded from the database.
27. **@OrderBy**: Defines the ordering of a collection of entities when retrieved from the database.
28. **@AttributeOverride**: Used to override the mapping of an embedded field in the entity.
29. **@AttributeOverrides**: Allows specifying multiple @AttributeOverride annotations for an embedded object.

**JUnit Annotations**

1. **@Test**: Marks a method as a test method, which will be executed by the JUnit test runner.
2. **@BeforeEach**: Indicates that the annotated method should be run before each test method in the current class.
3. **@AfterEach**: Indicates that the annotated method should be run after each test method in the current class.
4. **@BeforeAll**: Indicates that the annotated method should be run once before all test methods in the current class. It must be static.
5. **@AfterAll**: Indicates that the annotated method should be run once after all test methods in the current class. It must be static.
6. **@DisplayName**: Provides a custom display name for the test class or method, enhancing readability in test reports.
7. **@Nested**: Allows you to group tests in a nested structure, enabling logical grouping and shared setup.
8. **@ParameterizedTest**: Marks a method as a parameterized test, allowing the same test to be executed with different parameters.
9. **@ValueSource**: Supplies a single argument to a parameterized test method.
10. **@MethodSource**: Supplies arguments to a parameterized test method from a method.
11. **@CsvSource**: Supplies a comma-separated list of values to a parameterized test method.
12. **@AssertEquals**: Used to assert that two values are equal (commonly used within a test method).
13. **@Assertions**: A utility class that provides static methods for asserting conditions in tests (e.g., Assertions.assertNotNull()).

**Mockito Annotations**

1. **@Mock**: Creates a mock instance of a class or interface. This instance can be used to stub method calls and verify interactions.
2. **@InjectMocks**: Creates an instance of the class and injects the mocks created with @Mock or @Spy into it. Useful for testing classes with dependencies.
3. **@Spy**: Creates a spy instance of a class, allowing you to call real methods while still being able to stub specific calls.
4. **@Captor**: Creates an argument captor for capturing arguments passed to mocked methods.
5. **@BeforeEach** (JUnit with Mockito): Can be used to set up mocks before each test method.
6. **@AfterEach** (JUnit with Mockito): Can be used to perform cleanup after each test method.
7. **@RunWith(MockitoJUnitRunner.class)**: (JUnit 4 only) Specifies that the test should use the Mockito JUnit runner, enabling support for Mockito annotations. In JUnit 5, this is replaced with MockitoExtension.
8. **@ExtendWith(MockitoExtension.class)**: (JUnit 5 only) Enables support for Mockito annotations in a test class.

**Common Mockito Methods (Not Annotations)**

1. **when()**: Stubs a method call on a mock object to return a specific value.
2. **thenReturn()**: Specifies what value to return when a stubbed method is called.
3. **verify()**: Verifies that a specific method was called on a mock object.
4. **times()**: Specifies how many times a method should have been called in the verify() method.
5. **verifyNoInteractions()**: Verifies that no methods were called on the mock.
6. **doNothing()**: Used to specify that a method should do nothing when called, typically used with spies.
7. **doThrow()**: Specifies that a method should throw an exception when called.
8. **argThat()**: Allows capturing arguments based on a custom condition in the verify() method.

**Spring Actuator Annotations**

1. **@Endpoint**:
   * Marks a class as a custom Spring Boot Actuator endpoint. This annotation is used to create a custom endpoint that can expose additional functionality.
2. **@ReadOperation**:
   * Indicates that a method in a custom endpoint can be invoked with a GET request. It is typically used to expose read-only data.
3. **@WriteOperation**:
   * Indicates that a method can be invoked with a POST request to perform a write operation. It’s used for operations that modify the application state.
4. **@DeleteOperation**:
   * Marks a method in a custom endpoint as handling DELETE requests. It's used for operations that delete resources.
5. **@EndpointWebExtension**:
   * Allows adding web extensions to existing actuator endpoints. It enables the addition of additional operations for endpoints that are already provided by Spring Actuator.
6. **@Component**:
   * Although not specific to Actuator, this annotation is often used in combination with Actuator annotations to define a bean that will be picked up by component scanning.

**Commonly Used Actuator Endpoints**

1. **/actuator/health**:
   * Provides information about the application's health status, such as whether the application is up or down.
2. **/actuator/info**:
   * Displays arbitrary application information, such as build version, description, and custom information defined in the application properties.
3. **/actuator/env**:
   * Exposes environment properties, allowing you to view configuration properties and their current values.
4. **/actuator/metrics**:
   * Provides metrics related to the application, such as memory usage, garbage collection, and more.
5. **/actuator/loggers**:
   * Allows for viewing and modifying the logging levels of the application.
6. **/actuator/threaddump**:
   * Provides a thread dump of the application, helping to analyze thread states.
7. **/actuator/httptrace**:
   * Displays HTTP trace information, such as request and response details, useful for monitoring and debugging.

**Spring Security Annotations**

1. **@EnableWebSecurity**: Enables Spring Security’s web security support.
2. **@Configuration**: Marks the class as a source of bean definitions (commonly used with @EnableWebSecurity for security configuration).
3. **@PreAuthorize**: Restricts access to methods based on the roles or permissions provided in the expression (e.g., @PreAuthorize("hasRole('ADMIN')")).
4. **@Secured**: Limits method access to users with specific roles (e.g., @Secured("ROLE\_ADMIN")).
5. **@RolesAllowed**: Similar to @Secured, defines which roles can access a method (from JSR-250).
6. **@EnableGlobalMethodSecurity**: Enables method-level security annotations like @PreAuthorize and @Secured.
7. **@PermitAll**: Allows unrestricted access to a method.
8. **@DenyAll**: Denies access to a method for all users.
9. **@AuthenticationPrincipal**: Accesses the currently authenticated user.
10. **@EnableOAuth2Sso**: Enables OAuth2 Single Sign-On (SSO) support in Spring Boot applications.
11. **@WithMockUser**: Used in testing to mock the presence of a user with specified roles and permissions.
12. **@EnableResourceServer**: Enables the resource server capabilities for OAuth2.
13. **@EnableAuthorizationServer**: Enables the authorization server for handling OAuth2 authentication.

**Spring Microservices Annotations**

1. **@EnableDiscoveryClient**: Enables service registration and discovery using tools like Eureka, Consul, or Zookeeper.
2. **@EnableEurekaClient**: Specifically enables Eureka service registration and discovery.
3. **@FeignClient**: Declares a REST client that automatically uses HTTP calls to a specified service.
4. **@LoadBalanced**: Enables client-side load balancing using Ribbon.
5. **@EnableCircuitBreaker**: Enables Circuit Breaker pattern for fault tolerance in microservices.
6. **@HystrixCommand**: Provides fallback methods for remote service calls in case of failure.
7. **@EnableZuulProxy**: Enables Zuul API Gateway, which acts as a reverse proxy to forward requests to other services.
8. **@RibbonClient**: Configures a Ribbon client for load balancing across microservices.
9. **@EnableFeignClients**: Enables the use of Feign clients for calling other microservices.
10. **@EnableConfigServer**: Enables the Spring Cloud Config Server to provide external configuration for services.
11. **@EnableHystrix**: Enables the Hystrix Circuit Breaker in the application.
12. **@EnableKafka**: Enables Kafka message-driven POJO support in the Spring application.
13. **@StreamListener**: Listens for messages on a specific inbound channel in a microservice.
14. **@SendTo**: Sends the result of a method to a specified outbound channel.
15. **@Retryable**: Marks a method as retryable in case of transient faults.
16. **@EnableBinding**: Binds input and output channels to a message broker like Kafka or RabbitMQ in Spring Cloud Stream.