spring Boot Reference Documentation:

<https://docs.spring.io/spring-boot/docs/current/reference/html/index.html>

spring Boot:

Spring Boot an open-source framework that allows to create stand-alone production-grade spring based runnable application.

List of Starters:

Spring offers a wide range of starters POMs that can be used in your application. Here is the list of started POMs mention in this Spring Boot Tutorial.

| **Name** | **Description** | **Pom** |
| --- | --- | --- |
| spring-boot-starter-test | Starter for testing Spring Boot applications with libraries including JUnit, Hamcrest and Mockito | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-test/pom.xml) |
| spring-boot-starter-mobile | Starter for building web applications using Spring Mobile | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-mobile/pom.xml) |
| spring-boot-starter-social-twitter | Starter for using Spring Social Twitter | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-social-twitter/pom.xml) |
| spring-boot-starter-cache | Starter for using Spring Framework’s caching support | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-cache/pom.xml) |
| spring-boot-starter-activemq | Starter for JMS messaging using Apache ActiveMQ | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-activemq/pom.xml) |
| spring-boot-starter-jta-atomikos | Starter for JTA transactions using Atomikos | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-jta-atomikos/pom.xml) |
| spring-boot-starter-aop | Starter for aspect-oriented programming with Spring AOP and AspectJ | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-aop/pom.xml) |
| spring-boot-starter-web | Starter for building web, including RESTful, applications using Spring MVC. Uses Tomcat as the default embedded container | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-web/pom.xml) |
| spring-boot-starter-data-elasticsearch | Starter for using Elasticsearch search and analytics engine and Spring Data Elasticsearch | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-elasticsearch/pom.xml) |
| spring-boot-starter-jdbc | Starter for using JDBC with the Tomcat JDBC connection pool | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-jdbc/pom.xml) |
| spring-boot-starter-batch | Starter for using Spring Batch, including HSQLDB in-memory database | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-batch/pom.xml) |
| spring-boot-starter-social-facebook | Starter for using Spring Social Facebook | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-social-facebook/pom.xml) |
| spring-boot-starter-web-services | Starter for using Spring Web Services | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-web-services/pom.xml) |
| spring-boot-starter-jta-narayana | Spring Boot Narayana JTA Starter | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-jta-narayana/pom.xml) |
| spring-boot-starter-thymeleaf | Starter for building MVC web applications using Thymeleaf views | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-thymeleaf/pom.xml) |
| spring-boot-starter-mail | Starter for using Java Mail and Spring Framework’s email sending support | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-mail/pom.xml) |
| spring-boot-starter-jta-bitronix | Starter for JTA transactions using Bitronix | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-jta-bitronix/pom.xml) |
| spring-boot-starter-data-mongodb | Starter for using MongoDB document-oriented database and Spring Data MongoDB | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-mongodb/pom.xml) |
| spring-boot-starter-validation | Starter for using Java Bean Validation with Hibernate Validator | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-validation/pom.xml) |
| spring-boot-starter-jooq | Starter for using jOOQ to access SQL databases. An alternative to [spring-boot-starter-data-jpa](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#spring-boot-starter-data-jpa) or [spring-boot-starter-jdbc](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#spring-boot-starter-jdbc) | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-jooq/pom.xml) |
| spring-boot-starter-redis | Starter for using Redis key-value data store with Spring Data Redis and the Jedis client. Deprecated as of 1.4 in favor of [spring-boot-starter-data-redis](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#spring-boot-starter-data-redis) | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-redis/pom.xml) |
| spring-boot-starter-data-cassandra | Starter for using Cassandra distributed database and Spring Data Cassandra | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-cassandra/pom.xml) |
| spring-boot-starter-hateoas | Starter for building hypermedia-based RESTful web application with Spring MVC and Spring HATEOAS | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-hateoas/pom.xml) |
| spring-boot-starter-integration | Starter for using Spring Integration | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-integration/pom.xml) |
| spring-boot-starter-data-solr | Starter for using the Apache Solr search platform with Spring Data Solr | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-solr/pom.xml) |
| spring-boot-starter-freemarker | Starter for building MVC web applications using Freemarker views | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-freemarker/pom.xml) |
| spring-boot-starter-jersey | Starter for building RESTful web applications using JAX-RS and Jersey. An alternative to [spring-boot-starter-web](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#spring-boot-starter-web) | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-jersey/pom.xml) |
| spring-boot-starter | Core starter, including auto-configuration support, logging and YAML | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter/pom.xml) |
| spring-boot-starter-data-couchbase | Starter for using Couchbase document-oriented database and Spring Data Couchbase | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-couchbase/pom.xml) |
| spring-boot-starter-artemis | Starter for JMS messaging using Apache Artemis | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-artemis/pom.xml) |
| spring-boot-starter-cloud-connectors | Starter for using Spring Cloud Connectors which simplifies connecting to services in cloud platforms like Cloud Foundry and Heroku | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-cloud-connectors/pom.xml) |
| spring-boot-starter-social-linkedin | Stater for using Spring Social LinkedIn | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-social-linkedin/pom.xml) |
| spring-boot-starter-velocity | Starter for building MVC web applications using Velocity views. Deprecated since 1.4 | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-velocity/pom.xml) |
| spring-boot-starter-data-rest | Starter for exposing Spring Data repositories over REST using Spring Data REST | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-rest/pom.xml) |
| spring-boot-starter-data-gemfire | Starter for using GemFire distributed data store and Spring Data GemFire | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-gemfire/pom.xml) |
| spring-boot-starter-groovy-templates | Starter for building MVC web applications using Groovy Templates views | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-groovy-templates/pom.xml) |
| spring-boot-starter-amqp | Starter for using Spring AMQP and Rabbit MQ | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-amqp/pom.xml) |
| spring-boot-starter-hornetq | Starter for JMS messaging using HornetQ. Deprecated as of 1.4 in favor of [spring-boot-starter-artemis](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#spring-boot-starter-artemis) | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-hornetq/pom.xml) |
| spring-boot-starter-ws | Starter for using Spring Web Services. Deprecated as of 1.4 in favor of [spring-boot-starter-web-services](http://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#spring-boot-starter-web-services) | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-ws/pom.xml) |
| spring-boot-starter-security | Starter for using Spring Security | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-security/pom.xml) |
| spring-boot-starter-data-redis | Starter for using Redis key-value data store with Spring Data Redis and the Jedis client | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-redis/pom.xml) |
| spring-boot-starter-websocket | Starter for building WebSocket applications using Spring Framework’s WebSocket support | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-websocket/pom.xml) |
| spring-boot-starter-mustache | Starter for building MVC web applications using Mustache views | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-mustache/pom.xml) |
| spring-boot-starter-data-neo4j | Starter for using Neo4j graph database and Spring Data Neo4j | [Pom](https://github.com/spring-projects/spring-boot/tree/master/spring-boot-starters/spring-boot-starter-data-neo4j/pom.xml) |
| spring-boot-starter-data-jpa | Starter for using Spring Data JPA with Hibernate |  |

**Starring Spring Boot:**

We annotate the main class with @SpringBootApplicatoin and we run the application by this line of code

SpringApplication.run(App.class, args);

**Starting spring project:**

Group id: is like a package for project, Artifact is like project name.

**When spring boot run:**

**It starts spring application context** (springs advanced container which can load bean definitions, wire beans together, and dispense beans upon request and so on. Application context includes all functionality of BeanFactory. BeanFactory is the same us application context, but used for lightweight application like mobile), **it starts class path scan** (it scans all @ and managing accordingly), **IT starts Tomcat server automatically** (web server to host java servlets)

**Spring Ioc Container:**

It represents the Spring IoC Container is the core of Spring Framework. It creates the **objects**, **configures** and **assembles their dependencies**, **manages their entire life cycle**. The Container uses Dependency Injection(DI) to manage the components that make up the application.

It gets the information about the objects from a configuration file(XML) or Java Code or Java Annotations and [Java POJO class](https://www.geeksforgeeks.org/pojo-vs-java-beans/). These objects are called **Beans**. Since the Controlling of Java objects and their lifecycle is not done by the developers, hence the name **Inversion Of Control** (**Ioc**).

**Two types of Spring IoC Containers**

* BeanFactory container
* ApplicationContext container

**Application Context:**

It represents the Spring IoC container and is responsible for instantiating, configuring, and assembling the

Beans.

**BeanFactory:**

It is the **root interface** to access any Spring bean container. Enables us to read bean definitions and access them using the bean factory.

**BeanFactory** provides basic functionality for managing and manipulating beans, often in a programmatic way, the **ApplicationContext** provides extra functionality like MessageSource, Access to resources, Event propagation to beans, Loading of multiple (hierarchical) contexts etc.

Graphical user interface, diagram, application

Description automatically generated

**Dependency Injection:**

Dependency Injection is a design pattern that allows the spring container to ‘inject’ objects into other objects or dependencies. Dependency injection is an approach to implement loose coupling among the classes in an application.

**We can implement dependency injection with:**

* constructor-based injection,
* setter-based injection, or
* field-based injection.

**Constructor Injection:**

In this approach, beans are immutable and dependencies are not null. However, if you define many parameters in the constructor, your code is not clean.

@Component

**public** **class** SecondBeanImpl **implements** SecondBean {

**private** FirstBean firstBean;

    @Autowired

**public** SecondBeanImpl(FirstBean firstBean) {

**this**.firstBean = firstBean;

    }

}

**Setter Injection:**

In this approach, beans are not immutable (the setter could be called later), and not mandatory dependencies can lead to NullPointerExceptions. It is a good rule of thumb to use constructors for mandatory dependencies and setter methods or configuration methods for optional dependencies.

many developers prefer to use constructor injection over setter based injection because this makes bean class object as immutable(We cannot change the value by Constructor injection).

@Component

**public** **class** SecondBeanImpl **implements** SecondBean {

**private** FirstBean firstBean;

    @Autowired

**public** setFirstBean(FirstBean firstBean) {

**this**.firstBean = firstBean;

    }

}

**Field Injection:**

This approach may look cleaner but hides the dependencies and makes testing difficult. While constructor and setter injections use proxies, field injection uses reflection which could affect the performance.

@Component

**public** **class** SecondBeanImpl **implements** SecondBean {

    @Autowired

**private** FirstBean firstBean;

}

**Autoconfiguration:**

Spring Boot auto-configuration attempts to automatically configure your Spring application based on the class path(pom.xml=>jar) dependencies that you have added. To Enable auto configuration, we add **@EnableAutoConfiguration** or **@SpringBootApplication** annotation to one of your **@Configuration classes. Ex**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

Since, *spring-boot-starter-web* dependency added to class path leads to configure ***Tomcat*** and ***Spring MVC***

**@EnableAutoConfiguration:**

Enables spring boot automatically configure spring application based on dependencies that we added on class path(pom.xml)

**@ComoonentScan:**

It allows us spring to scan all components (**@Component, @Service, @Repository, @Controller** etc.) inside the package and automatically discovered and registered as spring bens in Springs’s application context.

we use the *@ComponentScan* annotation along with the *@Configuration* annotation to specify the packages that we want to be scanned.

@Configuration

@ComponentScan(basePackages= "io.reflectoring.vehicles") // or we use it without argument to scan the current package

public class ExplicitScan {

……………….

}

**@Configuration:**

is a class-based annotation that indicates the definition of one or more Bean methods in the class.

@Configuration

public class ConfigClass {

@Bean

public MyBean mybean() {

return new MyBean();

}

}

**@SpringBootApplicatoin:**

**@Configuration**: Allows the developers to explicitly register the beans or define bean

**@ComponentScan**: Enables the component-scanning so that the controller class and other components will be

automatically discovered and registered as beans in spring’s application context

**@EnableAutoConfiguration**: Enables spring boot automatically configure spring application based on

dependencies

**@Autowired:**

Autowiring feature of spring framework enables you to inject the object dependency implicitly. the spring container auto-wires the bean by matching data-type.

**@Constructo injection vs Setter:**

Constructor injection is mandatory whereas setter is not

**@Componenet**

Is a class level annotation. It allows spring to automatically detect our custom beans when we use annotation based configuration.

* **@Component** should be used when your class does not fall into either of three categories i.e. ***Controllers***, ***Services***and ***DAOs***.

Diagram

Description automatically generated

**@Bean:**

Is a method-level annotation. It tells the method to produce a bean to be managed by Spring Container.

**@Bean vs Component:**

*@Component* is a class-level annotation, but *@Bean* is at the method level, *@Component* is compatible with

Spring's auto-detection, but *@Bean* requires manual class instantiation.

**@Contrtoller:**

Is a specialization of the stereotype @Component annotation. @Controller annotation works with **@ResponseBody** annotation, and both annotations mark the class as a controller where every method returns a domain object instead of view.

**@Service:**

Is a specialization of the stereotype @Component annotation and we annotate the class to indicate

that the class is bean of service type. And service class communicate with Data Access Layer. (Spring context will autodetect these classes when annotation-based configuration and class path scanning is used)

**@Repository**

Is a specialization of the stereotype @Component annotation and we annotate the class to indicate that the class is bean of repository or DAO(data access object) type. which can be used as a mechanism for encapsulating storage, retrieval, and search a collection of Objects.

**@Restcontroller:**

This annotation used at the class level, and it marks the class as a controller where every method returns a

domain object instead of view. IT combines bellow annotations **@Controller and @ResponseBody.**

**@GetMapping:**

 It maps the **HTTP GET** requests on the specific handler method. It is used to create a web service

endpoint that **fetches** It is used instead of using: **@RequestMapping(method = RequestMethod.GET)**

**@PostMapping:**

It maps the **HTTP POST**requests on the specific handler method. It is used to create a web service

endpoint that **creates** It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**

**@PutMapping:**

It maps the **HTTP PUT** requests on the specific handler method. It is used to create a web service

endpoint that **creates** or **updates** It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**

**@DeleteMapping:**

It maps the **HTTP DELETE** requests on the specific handler method. It is used to create a web

service endpoint that **deletes**a resource. It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**

**@PatchMapping:**

It maps the **HTTP PATCH**requests on the specific handler method. It is used instead of

using: **@RequestMapping(method = RequestMethod.PATCH)**

**@RequestBody:**

It is used to **bind** HTTP request with an object in a method parameter. Internally it uses **HTTP**

**MessageConverters** to convert the body of the request. When we annotate a method parameter

with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.

**@ResponseBody:**

It binds the method return value to the response body. It tells the Spring Boot Framework to

serialize a return an object into JSON and XML format.

**@PathVariable:**

It is used to extract the values from the URI. It is most suitable for the RESTful web service, where

the URL contains a path variable. We can define multiple @PathVariable in a method.

**@RequestParam:**

 It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It

is most suitable for web applications. It can specify default values if the query parameter is not present in the URL.

**@RequestHeader:**

It is used to get the details about the HTTP request headers. We use this annotation as a **method**

**parameter**. The optional elements of the annotation are **name, required, value, defaultValue.**For each detail in the header, we should specify separate annotations. We can use it multiple time in a method

**@ResponseStatus:**

allows us to modify the HTTP status of our response. It can be applied in the following places:

On the exception class itself, along with the **@ExceptionHandler** annotation on methods, and along with the **@ControllerAdvice** annotation on classes.

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public class NoSuchElementFoundException extends RuntimeException {

...

}

or

public NoSuchElementFoundException(String message){

super(HttpStatus.NOT\_FOUND, message);

}

**@ControllerAdvice:**

Allows us to address exception handling across the whole application. It can handle exceptions thrown from any place in your application.

@ControllerAdvice

class CustomControllerAdvice {

…………

}

**@ExceptionHandler:**

 It gives us a lot of flexibility in terms of handling exceptions

@ExceptionHandler

public ResponseEntity<String> handleNoSuchElementFoundException(NoSuchElementFoundException exception)

**@RestController:**

It can be considered as a combination of **@Controller** and **@ResponseBody**annotations**.**  It

eliminates the need for annotating each method with **@ResponseBody**.

**@RequestAttribute:**

It binds a method parameter to request attribute. It provides convenient access to the request

attributes from a controller method. With the help of **@RequestAttribute** annotation, we can access objects that are populated on the server-side.

**@Transactional**

By default when we use spring JPA interface it include @Transactional, so we don’t have to annotate the service class with @Transactional.

**@RequestMapping:**

It is used to map web requests onto specific handler classes and/or handler methods. It can be applied to the controller class to create the base URI or method.

@Controller @Controller

@RequestMapping("/home") public class HomeController(“/home”) // Default is GetMethod, to change it to

public class HomeController { @RequestMapping("/home"). // Post we use method = POST

}

**@Required**

Is a method-level annotation applied to the setter method of a bean property and thus making the setter-injection mandatory.

 @Required

**public** **void** setCid(Integer cid) {

**this**.cid = cid;

    }

**@Lazy**

To initialize custom beans lazily we annotate the bean method or class with @Lazy.Lazy Initialization is the concept of deferring object creation until the object is actually first used.

**@Value**

It is commonly used for injecting values into configuration variables

@Value("John"). Application.properties => car.brand=Audi

private String trainee; or

@Value("${car.brand")

private String brand;

**@Qualifier**

It allows spring to avoid ambiguity when it finds multiple beans of the same type during auto wiring injection. If the multiple bean return type(interface which be implemented is the same, spring ambiguity will happen

Diagram

Description automatically generatedDiagram

Description automatically generated

**@Primary**

It allows spring to avoid ambiguity when it finds multiple beans of the same type by giving first priority to the beans during auto wiring injection

@Configuration

**public** **class** **Config** {

@Bean // We inject the Employee interface tony tope will return

**public** Employee **johnEmployee**() { @Autowired

**return** **new** **Employee**("John"); private Employee emolpye;

}

@Bean

@Primary

**public** Employee **tonyEmployee**() {

**return** **new** **Employee**("Tony");

}

}

**Benefit of Spring Boot:**

* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and Configuration.
* It is very easy to integrate Spring Boot Application with Spring JDBC, Spring Data, Spring Security etc.
* It provides Embedded HTTP(web) servers like Tomcat, Jetty etc. to develop and test web applications very easily.
* It provides CLI (Command Line Interface) tool to develop and test Spring Boot (Java or Groovy) Applications from command prompt very easily and quickly.
* It provides lots of plugins to develop and test Spring Boot Applications very easily using Build Tools like Maven and Gradle

**HTTP Response Status Codes**

* **200 Everything is OK:**

This is the code that is delivered when a web page or resource acts exactly the way it’s expected to.

* **201 Created:**

This code indicates that the request is successful, and a new resource is created.

* **202 Accepted:**

The server has accepted your browser’s request but is still processing it. The request ultimately may or may not result in a completed response.

* **400 Bad Request:**

This code indicates that the server failed to process the request because of the malformed syntax in the request. The client can try again after correcting the request.

* **401 Unauthorized:**

This code indicates that authentication is required for the resource. The client can try again with appropriate authentication.

* **403 Forbidden:**

This code indicates that the server is refusing to respond to the request even if the request is valid. The reason will be listed in the body content if the request is not a HEAD method.

* **404 Not Found:**

This code indicates that the requested resource is not found at the location specified in the request.

* **500 Internal Server Error:**

This code indicates a generic error message, and it tells that an unexpected error occurred on the server and that the request cannot be fulfilled.

* **502:** **Bad Gateway:**

This error code typically means that one server has received an invalid response from another.

**Exception Handling**

JVM will

**@ResponseStatus:**

Allows us to modify the HTTP status of our response. It can be applied in the following places:

* On the exception class itself
* Along with the **@ExceptionHandler** annotation on methods
* Along with the **@ControllerAdvice** annotation on classes

@ResponseStatus(value = HttpStatus.NOT\_FOUND)

public class NoSuchElementFoundException extends RuntimeException {

Or

public NoSuchElementFoundException(String message){

super(HttpStatus.NOT\_FOUND, message);

}

**@ExceptionHandler**

Is used for handling exceptions in specific handler classes and/or handler methods.

@ExceptionHandler(NoSuchElementFoundException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public ResponseEntity<String> handleNoSuchElementFoundException(NoSuchElementFoundException exception ) {

………..

**@ControllerAdvice**

Allow us to apply exception handlers to more than one or all controllers in our application:

@ControllerAdvice

public class GlobalExceptionHandler extends ResponseEntityExceptionHandler {

**@ResponseStatus:**

JVM will

**@ResponseStatus:**

JVM will

**Don’t close resources manually:**

The JVM will close the resource for you when the try…catch block completes, eliminating the potential for messy and difficult to troubleshoot resource leaks.

public class TryWithResourcesExample {

public static void main(String[] args) throws Exception {

try (Door door = new Door()) {

door.swing();

} catch (Exception e) { /\* do something \*/ }

} finally { /\* do something \*/ }

/\* The door will automatically be closed after the code block \*/

}

}

Or

public void closeResourceInFinally() {

FileInputStream inputStream = null;

try {

File file = new File("./tmp.txt");

inputStream = new FileInputStream(file);

// use the inputStream to read a file

} catch (FileNotFoundException e) {

log.error(e);

} finally {

if (inputStream != null) {

try {

inputStream.close();

} catch (IOException e) {

log.error(e);

}

}

}

}

**Be careful What you Log:**

It is not a good practice to log protected data. Don’t catch exception and then do nothing with it. That’s known as burying an exception. At the very least log the name of the exception and the message associated with it.

**Use a Global Exception Handler:**

There will always be instances of the uncaught RuntimeException that will creep into your code. Always include a global Exception handler to deal with any uncaught exceptions.

**Throw early and handle exceptions late**

The As soon as an exception condition happens in your code, throw an Exception. Don’t wait for any additional lines of code to run before terminating execution of the method you are in.

**Don’t log and rethrow**

Doing so causes code duplication, and it also litters the log files with duplicate entries

/\* log and rethrow exception example \*/

**try** {

Class.*forName*("com.mcnz.Example");

} **catch** (ClassNotFoundException ex) {

*log*.warning("Class was not found.");

**throw** ex;

}

**Catch the Most Specific Exception First**

The first catch block handles all *NumberFormatException*s and the second one all *IllegalArgumentException*s which are not a *NumberFormatException*.

public void catchMostSpecificExceptionFirst() {

try {

doSomething("A message");

} catch (NumberFormatException e) {

log.error(e);

} catch (IllegalArgumentException e) {

log.error(e)

}

}

**Don’t Catch Throwable**

[*Throwable*](https://docs.oracle.com/javase/8/docs/api/java/lang/Throwable.html)is the superclass of all exceptions and errors. You can use it in a catch clause, but you should never do it! If you use ***Throwable*** in a catch clause, it will not only catch all exceptions; it will also catch all errors.

**Errors**:

are thrown by the JVM to indicate serious problems that are not intended to be handled by an application. Typical examples for that are the *[OutOfMemoryError](https://docs.oracle.com/javase/8/docs/api/java/lang/OutOfMemoryError.html)*or the *[StackOverflowError](https://docs.oracle.com/javase/8/docs/api/java/lang/StackOverflowError.html)*. Both are caused by situations that are outside of the control of the application and can’t be handled.

public void doNotCatchThrowable() {

try {

// do something

} catch (Throwable t) {

// don't do this!

}

}

**Don’t Ignore Exceptions:**

public void doNotIgnoreExceptions() {

try {

// do something

} catch (NumberFormatException e) {

// this will never happen

}

}

Instead add **log.error(“This should never happen: “ + e);** inside catch block.

**Global Exception Handler Sample Snippet**

Doing so causes

**ErrorDetails Pojo Class:**

**package** com.dailycodebuffer.user.exception;

// We add @JsonInclude(JsonInclude.Include.NON\_NULL)

// when jacson build this object, it only build not null object, so validationError {} will not

// Display when there is no error.

@JsonInclude(JsonInclude.Include.NON\_NULL)

**public** **class** ErrorDetails {

**private** Date date;

**private** String message;

**private** String details;

**private** Map<String , String> validationError;

**public** ErrorDetails(Date date, String message, String details) {

**this**.date = date;

**this**.message = message;

**this**.details = details;

}

……geter and setter method need to emplimented

**CustomerNotFoundException Custom Exception Class:**

**package** com.dailycodebuffer.user.exception;

**public** **class** CustomerNotFoundException **extends** RuntimeException{

**private** **static** **final** **long** serialVersionUID = 1L;

**public** CustomerNotFoundException(String message)

{

**super**(message);

}

}

**APIException Custom Exception Class:**

**package** com.dailycodebuffer.user.exception;

**public** **class** APIException **extends** RuntimeException {

**private** **static** **final** **long** serialVersionUID = 1L;

**public** APIException(String message){

**super**(message);

}

}

**GlobalExceptionHandler All Exception Handler Class:**

@ControllerAdvice

**public** **class** GlobalExceptionHandler {

// Handel Specific Exception

@ExceptionHandler(APIException.**class**)

**public** ResponseEntity<?> handleAPIException(APIException exception, WebRequest request){

ErrorDetails errorDetails = **new** ErrorDetails(**new** Date(), exception.getMessage(),

request.getDescription(**false**));

**return** **new** ResponseEntity<>(errorDetails, HttpStatus.NOT\_FOUND);

}

// Handel Specific Exception Similar Implimentation

@ExceptionHandler(CustomerNotFoundException.**class**)

**public** ResponseEntity<?> handleCustomerNotFoundExceptin(CustomerNotFoundException exception,

HttpServletRequest httpServletRequest){

ErrorDetails errorDetails = **new** ErrorDetails(**new** Date(), exception.getMessage(),

httpServletRequest.getServletPath());

**return** **new** ResponseEntity<>(errorDetails, HttpStatus.NOT\_FOUND);

}

// This handle MethodArgumentNotValidException during post request.

@ExceptionHandler(MethodArgumentNotValidException.**class**)

**public** ResponseEntity<?> handleMethodArgumentNotValidException(MethodArgumentNotValidException

exception, HttpServletRequest httpServletRequest){

ErrorDetails errorDetails = **new** ErrorDetails(**new** Date(), "Bad request",

httpServletRequest.getServletPath());

//It gets all the binding result with multiple errors

BindingResult bindingResult = exception.getBindingResult();

Map<String, String> validationErrors = **new** HashMap<>();

// loop on multiple Field Errors

**for**(FieldError fieldError : bindingResult.getFieldErrors()){

validationErrors.put(fieldError.getField(), fieldError.getDefaultMessage());

}

errorDetails.setValidationError(validationErrors);

**return** **new** ResponseEntity<>(errorDetails, HttpStatus.BAD\_REQUEST);

}

// Handel Global Exception

@ExceptionHandler(Exception.**class**)

**public** ResponseEntity<?> handleException(Exception exception, WebRequest request){

ErrorDetails errorDetails = **new** ErrorDetails(**new** Date(), exception.getMessage(),

request.getDescription(**false**));

**return** **new** ResponseEntity<>(errorDetails, HttpStatus.INTERNAL\_SERVER\_ERROR);

}

}

**Testing**

**Unit test:**

A test which tests a functionality of class in isolated way by mocking the dependency.

Reference: <https://junit.org/junit5/docs/current/user-guide/#overview>

**Junit 5:**

JUnit 5 = *JUnit Platform* + *JUnit Jupiter* + *JUnit Vintage*

Diagram

Description automatically generated

We need bellow dependency to work in newer Junit 5:

<!--Junit 5 Dependency-->  
 **<dependency>**  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <version>${junit.jupiter.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
  **<dependency>**  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <version>${junit.jupiter.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
 **<dependency>**  
 <groupId>org.junit.platform</groupId>  
 <artifactId>junit-platform-suite</artifactId>  
 <version>${junit.platform.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
<!-- End -->

**JUnit Jupiter:**

Is the combination of the [programming model](https://junit.org/junit5/docs/current/user-guide/#writing-tests) and [extension model](https://junit.org/junit5/docs/current/user-guide/#extensions) for writing tests and extensions in JUnit 5. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.

**JUnit Vintage:**

provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform. It requires JUnit 4.12 or later to be present on the class path or module path.

**JUnit 4:**

It is 10 years old monolithic architecture which only has one jar. It is not updated with newer java version.

**Test Lifecyle Phases:**

Create a new instance of the test class before running each test method.

**Graphical user interface

Description automatically generated**

**@BeforeAll and @AfterAll:** annotations – clear by their name – should be called only once in the entire tests execution

cycle. So they must be declared **static** because the test class will not be initialized since we annotate the method with @beforeall or @afterAll. Therefore, the method will not have class to run. So, we have to make the method static because static method can be runed before the class instance created.

**@ BeforeAll @AfterAll**

**public** static Employee **johnEmployee**() { **public** static Employee **johnEmployee**() {

**@BeforeEach and @AfterEach:** are invoked for each instance of test so they should not be static.

@BeforeAll executed

@BeforeEach executed

======TEST ONE EXECUTED=======

@AfterEach executed

@BeforeEach executed

======TEST TWO EXECUTED=======

@AfterEach executed

@AfterAll executed

**@TestInstance:**

To check the behavior of test instance (by default one instance created per method test) to one instance per class, we use **@TestInstance**(TestInstance.Lifecycle.PER\_CLASS)

Class TestClass{

If we create on instance per test class, we can run **@BeforeAll and @AfterAll** without static method.

**@DisplayName(“name”):**

To show custom method name while checking the console output.

**@Test**

**@DisplayName(“Testing add method”)**

**void** **addNumbers()** **{**

**@ExtendWith(MockitoExtension.class):**

To load a Junit5 extension. It allows third-party vendor like Mockito to hook into the lifecycle of running test classes and add additional functionality. *The MockitoExtension looks at the test class, finds member variables annotated with the @Mock annotation, and creates a mock implementation of those variables. It then finds member variables annotated with the @InjectMocks annotation and attempts to inject its mocks into those classes, using either construction injection or setter injection*.

**@TxtendWith(MockitoExtension.class)**

Class TestClass{

***@Mock***

***Repository repository;***

***@InjectMocks***

***Service service;***

**@Test**

**Final void testGetMethod(){**

**…………….**

**}**

**}**

If we don’t use Junit5 extension **@TxtendWith(MockitoExtension.class)** at class level we need to setup the Mock annotation before each test as below.

**Class TestClass {**

***@Mock***

***Repository repository;***

***@InjectMocks***

***Service service;***

**@BeforeEach**

**void setUp() throws Exception {**

**MockitoAnnotations.initMocks(this);**

**}**

**@Test**

**Final void testGetMethod(){**

**…………….**

**}**

**}**

**@Disabled:**

To Disable test method.

**@Test**

**@Disabled**

**void** **addNumbers()** **{**

**@Tag(“name”):**

To distinguish important tests. So, that we chose the important test when we run the tests.

**@Test**

**@Tag(“Adition “)**

**void** **addNumbers()** **{**

**@RepeatedTest(n):**

It repletes test n times.

**@Test**

**@RepetedTest(4)**

**void** **addNumbers()** **{**

**@Netsted:**

It groups multiple tests inside test class.

**@Nested**

**class AddTest {**

**@Test**

**void testAddingTwoPositives() {**

**assertEquals(2, mathUtils.add(1, 1), "Add method should return the sum of two numbers");**

**}**

**@Test**

**void testAddingTwoNegatives() {**

**assertEquals(-2, mathUtils.add(-1, -1), "Add method should return the sum of two numbers");**

**}**

**@Test**

**void testAddingAPositiveAndANegative() {**

**assertEquals(0, mathUtils.add(-1, 1), "Add method should return the sum of two numbers");**

**}**

**}**

**Assertions**

JUnit Jupiter comes with many of the assertion methods that JUnit 4 has and adds a few that lend themselves

well to being used with Java 8 lambdas. All JUnit Jupiter assertions are static methods in the.

**AssertAll:**

It runs multiple assertions in one statement.

**@Test**

**@DisplayName(“Testing multiply method”)**

**void testMultiply() {**

**assertAll(**

**() -> assertEquals(0, mathUtils.multiply(1, 0)),**

**() -> assertEquals(1, mathUtils.multiply(1, 1)),**

**() -> assertEquals(6, mathUtils.multiply(2, 3))**

**);**

**}**

**Value Test:**

**assertEquals() and assertNotEquals()**

**@Test**

**void** **addNumbers()** **{**

Calculator calculator **=** **new** Calculator**();**

assertEquals**(**3**,** calculator**.**add**(**1**,** 2**), “The add method should add tow numbers”);**

**}**

If the assertion failed, we would see both the expected and actual values in the error message:

Expected :3

Actual :2

**Boolean Test:**

**assertTrue() and assertFalse()**

**@Test**

**void** **firstNameStartsWithJ()** **{**

Person person **=** **new** Person**(**"John"**,** "Doe"**);**

assertTrue**(**person**.**getFirstName**().**startsWith**(**"J"**));**

**}**

**Null Test:**

**assertNull() and assertNotNull()** to test object is null or not

**@Test**

**void** **personHasFirstName()** **{**

Person person **=** **new** Person**(**"John"**,** "Doe"**);**

assertNotNull**(**person**.**getFirstName**());**

**}**

**Collection Item Test:**

**assertIterableEquals()**

**@Test**

**void** **iterablesEqual()** **{**

**final** List**<**String**>** list **=** Arrays**.**asList**(**"orange"**,** "mango"**,** "banana"**);**

**final** List**<**String**>** expected **=** Arrays**.**asList**(**"banana"**,** "mango"**,** "orange"**);**

Collections**.**sort**(**list**);**

assertIterableEquals**(**expected**,** list**);**

**}**

**Arrays Test:**

**assertArrayEquals()**

**@Test**

**void** **arraysEqual()** **{**

**final** **int[]** array **=** **{** 3**,** 2**,** 1 **};**

**final** **int[]** expected **=** **{** 1**,** 2**,** 3 **};**

Arrays**.**sort**(**array**);**

assertArrayEquals**(**expected**,** array**);**

**}**

**Object Test:**

**assertEquals() and assertNotEquals()**

**@Test**

**void** **personsAreSame()** **{**

Person john **=** **new** Person**(**"John"**,** "Doe"**);**

Person doe **=** **new** Person**(**"John"**,** "Doe"**);**

assertEquals**(**john**,** doe**);**

**}**

This will fail because Java object equality uses the **equals()** method for comparison. The default implementation of the **equals()** method checks if two object references refer to the same object .So, we have to override **equals()**. If we override the **equals()** method, we also have to override **hashCode().**

**public** **class** **Person** **{**

*// ...*

**@Override**

**public** **boolean** **equals(**Object o**)** **{**

**if** **(this** **==** o**)** **return** **true;**

**if** **(**o **==** **null** **||** getClass**()** **!=** o**.**getClass**())** **return** **false;**

Person person **=** **(**Person**)** o**;**

**return** firstName**.**equals**(**person**.**firstName**)** **&&** lastName**.**equals**(**person**.**lastName**);**

**}**

**@Override**

**public** **int** **hashCode()** **{**

**return** Objects**.**hash**(**firstName**,** lastName**);**

**}**

**}**

**Exceptions Test:**

**assertThrows()**

**@Test**

**void** **divideByZeroThrowsIllegalArgumentException()** **{**

Calculator calculator **=** **new** Calculator**();**

assertThrows**(IllegalArgumentException.class ,** **()** **->** calculator**.**divide**(**1**,** 0**));**

**}**

In this example, the implementation will throw **IllegalArgumentException** if we try to divide by zero.

If it doesn’t throw an exception, the test will fail with an error message:

**To verify information about the exception:**

**@Test**

**void** **divideByZeroThrowsIllegalArgumentException()** **{**

Calculator calculator **=** **new** Calculator**();**

Throwable thrown **=** assertThrows**(**IllegalArgumentException**.**class**,** **()** **->** calculator**.**divide**(**1**,** 0**));**

assertEquals**(**"Cannot divide by zero"**,** thrown**.**getMessage**());**

**}**

**Timeouts Test:**

**assertTimeout() or the assertTimeoutPreemptively()**

The difference between these two methods is that **assertTimeout()** runs in the same thread as the code that calls it, and it won’t abort if it exceeds the timeout. On the other hand, the **assertTimeoutPreemptively()** method executes in a different thread and aborts if it exceeds the timeout.

**@Test**

**void** **returnValueBeforeTimeoutExceeded()** **{**

**final** String message **=** assertTimeout**(**Duration**.**ofMillis**(**50**),** **()** **->** **{**

Thread**.**sleep**(**100**);**

**return** "a message"**;**

**});**

assertEquals**(**"a message"**,** message**);**

**}**

Since the execution time will exceed the timeout, we will see an error message:

org.opentest4j.AssertionFailedError: execution exceeded timeout of 100 ms by 50 ms

**@Test**

**void** **abortWhenTimeoutExceeded()** **{**

**final** String message **=** assertTimeoutPreemptively**(**Duration**.**ofMillis**(**50**),** **()** **->** **{**

Thread**.**sleep**(**100**);**

**return** "another message"**;**

**});**

assertEquals**(**"another message"**,** message**);**

**}**

org.opentest4j.AssertionFailedError: execution timed out after 50 ms

The difference here is that the execution stopped at the timeout value.

**Arrays Test:**

**assertArrayEquals()**

**@Test**

**void** **arraysEqual()** **{**

**final** **int[]** array **=** **{** 3**,** 2**,** 1 **};**

**final** **int[]** expected **=** **{** 1**,** 2**,** 3 **};**

Arrays**.**sort**(**array**);**

assertArrayEquals**(**expected**,** array**);**

**Sample Junit 5 Test Cases:**

**@ExtendWith(SpringExtension.class)  
public class CustomerServiceTest {  
  
 @Mock  
 CustomerRepository customerRepository;  
  
 @InjectMocks  
 CustomerService customerService;  
  
  
 @Test  
 @DisplayName("Testing CustomerService class save method")  
 public void TestSaveCustomerMethod(){**

**Customer expected = new Customer(1L, "Ermi","Ted","test@gmail.com",2L);  
 Customer expected2 = new Customer(3L, "Tedi","Jhon","jhon@gmail.com",4L);  
  
 // Mockito.doNothing().when(customerRepository.save(Mockito.any(Customer.class)));  
 Mockito.*when*(customerRepository.save(Mockito.*any*(Customer.class))).thenReturn(expected);  
  
 Customer actual = customerService.saveCustomer(expected);  
 Assertions.*assertNotEquals*(expected2.getFirstName(), actual.getFirstName());  
 Assertions.*assertEquals*(expected.getFirstName(), actual.getFirstName());  
 }  
  
 @Test  
 @DisplayName("Testing CustomerService class get customers method")  
 public void TestGetCustomersMethod(){**

**Customer expected = new Customer(1L, "Ermi","Ted","test@gmail.com",2L);  
 Customer expected2 = new Customer(3L, "Tedi","Jhon","jhon@gmail.com",4L);  
 List<Customer> expectedCustomers = new ArrayList<>();  
 expectedCustomers.add(expected);  
 expectedCustomers.add(expected2);  
  
 Mockito.*when*(customerRepository.findAll()).thenReturn(expectedCustomers);  
  
 List<Customer> actual = customerService.getCustomers();  
  
 Assertions.*assertEquals*(expectedCustomers, actual);  
 Assertions.*assertEquals*(expectedCustomers.get(0).getFirstName(), actual.get(0).getFirstName());  
 }  
  
 @Test  
 @DisplayName("Testing CustomerService class get customers method exception")  
 public void TestGetCustomersMethod\_CustomerNotFoundException(){  
  
 Mockito.*when*(customerRepository.findAll()).thenReturn(Collections.*emptyList*());  
 Assertions.*assertThrows*(CustomerNotFoundException.class, () -> customerService.getCustomers());  
 }  
  
  
 @Test  
 @DisplayName("Testing CustomerService class get specific customer method")  
 public void TestGetSpecificCustomerMethod(){  
 Customer expected = new Customer(1L, "Ermi","Ted","test@gmail.com",2L);  
 Customer expected2 = new Customer(3L, "Tedi","Jhon","jhon@gmail.com",4L);  
  
 Mockito.*when*(customerRepository.findByCustomerId(Mockito.*anyLong*())).thenReturn(expected);  
 Customer actual = customerService.getCustomerById(1L);  
  
 Assertions.*assertEquals*(expected, actual);  
 Assertions.*assertEquals*(expected.getFirstName(), actual.getFirstName());  
 }  
 }**

**Dependency needed for Junit 5**

<!--Junit 5 Dependency-->  
 **<dependency>**  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <version>${junit.jupiter.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
  **<dependency>**  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <version>${junit.jupiter.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
 **<dependency>**  
 <groupId>org.junit.platform</groupId>  
 <artifactId>junit-platform-suite</artifactId>  
 <version>${junit.platform.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
<!-- End -->

**Mockito**

**Mockito:**

A java-based framework used for unit testing of java application. Mocking is a testing technique where mock objects are used instead of real objects for testing purposes.

**@Mock:**

**@Mock** annotation only creates mock or fake object.

**@MockBean**

**@MockBean** is spring boot annotation and it creates mock and **injects it into** Application Context.

Main difference between **@MockBean** and **@Mock** annotation is that **@MockBean** creates mock and **injects it into** Application Context, while **@Mock** annotation only **creates it**, if you want to inject it, you can do it manually or with **@InjectMocks** annotation

**@InjectMock:**

To create class instances that need to be tested in the test class. We call it ‘*code under test*‘ or ‘*system under test*‘. We will not mock service class because it is the calls, we need to test the functionality of the business logic.

***@Mock***

***Repository repository;***

***@InjectMocks***

***Service service;***

**Verify():**

**Verify()** in Mockito simply means that you want to check if a certain method of a mock object has been called by specific number of times.

**@Test**

**public void testMockListAddMultiple() {**

**String addString = "some string multiple";**

**mockList.add(addString);**

**mockList.add(addString);**

**mockList.add(addString);**

**//verify that the add method was called with argument 'some string'**

**verify(mockList, times(3)).add(addString);**

**}**

**Stubing:**

A *stub* is an object that always returns the same value, regardless of which parameters you provide on a stub’s methods.

**anyBoolean(), anyByte(), anyInt(), anyString(), anyDouble(), anyFloat(), anyList() and** many more – All of these accept any object of the corresponding data type as well as null values.

**Integration Test**

**Integration test:**

A test which communicates with network or database or file system and run.

**@ExtendWith:**

In spring to do integration test, unit test needs to get access to application context so that we can access to beans and property files. So, we use **@ExtendWith.**

**@RunWith(SpringRunner.class) @ExtendWith(SpringExtention.class)**

**@SpringBootTest replaced by this => @SpringBootTest**

**Class** **unitTest** **{ Junit 5 Class** **unitTest** **{**

**…..// …..//**

**}** **}**

Integration test:

Integration test can be categorized in two parts.

1. **Narrow integration tests** that exercise only part of the application and use test doubles for some components or external services. Some call these **component tests** or **service tests** to make the distinction.
2. **Broad integration tests** that need the whole application running and exercise the application through UI or network calls. Some call these **system tests** or **end-to-end tests** to make the distinction.

MockMvc:

It allows us to call Rest API methods.

@AutoConfiguerMockMvc:

To enable the **MockMvc**, we annotate the test class with **@AutoConfiguerMockMvc.**

**@AutoConfiguerMockMvc**

**@SpringBootTest**

**public class controllerIntegrationTest {**

**@Autowired**

**Private CustomerRepository customerRepository**

**@Autowired**

**privage MockMvc mockMvc**

………………

**}**

**Integration Test Sample**:

It allows us

@ExtendWith(SpringExtension.**class**)

@SpringBootTest

@AutoConfigureMockMvc

**public** **class** CustomerServiceControlerTest {

@Autowired

**private** MockMvc mockMvc;

@MockBean

CustomerRepository customerRepository;

@Test

**public** **void** testGetCustomerByCustomerId() **throws** Exception {

Long id = 1L;

Customer expected = **new** Customer(id, "Ermi", "Ted", "test@gmail.com", 2L);

Mockito.when(customerRepository.findByCustomerId(id)).thenReturn(expected);

mockMvc.perform(get("/customer/getCustomer/{id}", id))

.andExpect(status().isOk())

.andExpect(jsonPath("$.customerId").value(id))

.andExpect(jsonPath("$.firstName").value(expected.getFirstName()))

.andExpect(jsonPath("$.lastName").value(expected.getLastName()))

.andExpect(jsonPath("$.email").value(expected.getEmail()))

.andExpect(jsonPath("$.departmentId").value(expected.getDepartmentId()))

.andDo(MockMvcResultHandlers.print());

}

@Test

**public** **void** testGetCustomerShouldReturnCustomerNotFound() **throws** Exception {

Long id = 1L;

Mockito.when(customerRepository.findByCustomerId(id)).thenReturn(**null**);

mockMvc.perform(get("/customer/getCustomer/{id}", id))

.andExpect(status().isNotFound())

.andDo(MockMvcResultHandlers.print());

}

}

**Dependency for Junit 5 and Integration Test**

<!--Junit 5 Dependency-->  
 **<dependency>**  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-engine</artifactId>  
 <version>${junit.jupiter.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
  **<dependency>**  
 <groupId>org.junit.jupiter</groupId>  
 <artifactId>junit-jupiter-api</artifactId>  
 <version>${junit.jupiter.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
 **<dependency>**  
 <groupId>org.junit.platform</groupId>  
 <artifactId>junit-platform-suite</artifactId>  
 <version>${junit.platform.version}</version>  
 <scope>test</scope>  
 **</dependency>**  
<!-- End -->

<!-- JsonPath expression and Hamcrest dependency for matchers-->

**<dependency>**  
 <groupId>org.hamcrest</groupId>  
 <artifactId>hamcrest-library</artifactId>  
 <scope>test</scope>  
**</dependency>**  
**<dependency>**  
 <groupId>com.jayway.jsonpath</groupId>  
 <artifactId>json-path</artifactId>  
 <scope>test</scope>  
**</dependency>**

<!-- End -->

**Maven**

**maven:**

A build automation and dependency management tool. It downloads dependencies in class path once we

add dependency in pom.xml file.

Commands:

**Maven Dependency Lifecycle:**

Maven follows a sequential order to execute the commands. Ex. If we run <**mvn test**> all above Test command will run automatically

**Diagram

Description automatically generated**

**=> Validate:** This step validates if the project structure is correct. For example – It checks if all the dependencies

have been downloaded and are available in the local repository.

**=> Compile:** It compiles the source code, converts .java files to .class & stores the classes in target/classes folder.

**=> Test:** It runs unit tests for the project.

**=> Package:** This step packages the compiled code in distributable format like JAR or WAR.

**=> Integration test:** It runs the integration tests for the project.

**=> Verify:** This step runs checks to verify that the project is valid and meets the quality standards.

**=> Install:** This step installs the packaged code to the local Maven repository.

**=> Deploy:** It copies the packaged code to the remote repository for sharing it with other developers.

**Maven Commands:**

* **mvn clean** => Cleans the project and removes all files generated by the previous build.
* **mvn compile** => Compiles source code of the project.
* **mvn test-compile** => Compiles the test source code.
* **mvn test** => Runs tests for the project.
* **mvn package** => Creates JAR or WAR file for the project to convert it into a distributable format.
* **mvn install** => Deploys the packaged JAR/ WAR file to the local repository (creates runnable jar or war file)
* **mvn deploy** => Copies the packaged JAR/ WAR file to the remote repository after compiling, running tests and building the project.
* **mvn clean + above command** **=>** target director deleted from previous andgenerate new.
* **mvn -X + above command** **=>** the maven step will run in debug mode.
* **mvn install -DskipTests** => avoid the test step while running install.

**Java -jar <jar path>** => it runs generated java spring project.

**To deploy or run spring project as war =>** change the pom dependency to war instead of jar.

**Java 8 Stream**

**Ref:** <https://howtodoinjava.com/java/stream/java-streams-by-examples/>

**Ref:** [**https://www.java2novice.com/\**](https://www.java2novice.com/\)

**Ref:** [**https://www.javaguides.net/2021/11/java-stream-terminal-operations-examples.html**](https://www.javaguides.net/2021/11/java-stream-terminal-operations-examples.html)

**mapToDouble, average(). Example**

**Functional Interface:**

* + An interface with only one abstract method. Optionally we can annotate functional interface with

@FunctionalInterface.

* + Lambda expression work with functional interface.
  + It can have any number of default(which can have method body) & static methods but only have one abstract method.
  + **Runnable, Comparable, ActionListener,** & Callable are converted into functional interface.
  + **Java** **SE** **8** has 4 main Functional Interfaces.

**1, Consumer**

**2, Predicate**

**3, Function**

**4, Supplier**

**Consumer:** Itaccepts one argument, but there is no return value.

@FunctionalInterface

public interface **Consumer**<T>{

void accept(T t);

}

Consumer<String> p = **s** -> System.out.println(**s.toUpperCase()**);

p.accept("comsumer");

**BiConsumer:** It accepts two arguments and returns nothing.

**Predicate:** It accept one argument, do some processing, and then return boolean.

**BiPredicate**: It accepts two arguments & returns Boolean.

**Function:** It accepts one argument and returns a value after the required processing.

**BiFunction:** It accepts two arguments & returns value after the required processing.

**Supplier:**Is represents an operation that takes no argument and returns a result, whose functional

method is get().

- It has **get()** , getAsInt(), getAsLong(), getAsDouble(), getAsBoolean(), **orElseGet()..**

System.out.println(list.stream().findAny().**orElseGet**(() -> “Hi viewers”));

**Lambda Expressions:**

* + A lambda expression is a short block of code which takes in parameters and returns a value.

**Zero Parameter:()** -> System.out.println("Zero parameter lambda");

**One Parameter:** **(p)** -> System.out.println("One parameter: " + **p**);

**Multiple Parameters:** **(p1, p2)** -> System.out.println("Multiple parameters: " + **p1** + ", " + **p2**);

**Method References:**

* + It is used to refer methods of functional interfaces.

|  |  |  |
| --- | --- | --- |
| **K Kindd** | **Syntax** | **Example** |
| Reference to a static method | Class::staticMethodName | String::valueOf |
| Reference to an instance method | object::instanceMethodName | x::toString |
| Reference to a constructor | ClassName::new | String::new |

|  |  |  |
| --- | --- | --- |
| **Kind** | **As Method Reference** | **As Lambda** |
| Reference to a static method | String::valueOf | (s) -> String.valueOf(s) |
| Reference to an instance method | x::toString | () -> "java2novice".toString() |
| Reference to a constructor | String::new | () -> new String() |

**Stream API:**

Java stream operations are divided into intermediate and terminal operations.

Diagram

Description automatically generated

An Only one Intermediate operation(**limit()**) and Five Terminal operations (**findFirst(), allMatch(), anyMatch(), findAny(), noneMatch()**) is short circuiting in Java, if when presented with infinite input, it may produce a finite stream as a result.

Graphical user interface, diagram, text, application

Description automatically generated

**Predicae:**

In Java 8, Predicate is a functional interface, which accepts an argument and returns a boolean.

Usually, it used to apply in a filter for a collection of objects.

**@FunctionalInterface**

**public** **interface** Predicate<T> {

**boolean** test(T t);

}

**test(T t):**

Predicate<Integer> greater\_than = x -> (x > 10);

System.out.println(greater\_than.test(11));

**isEqual(Object t):**

Predicate<String> pred = Predicate.isEqual("Educative");

System.out.println(pred.test("educative "));

**and(Predicate P):**

Predicate<Integer> grt\_10 = x -> (x > 10);

Predicate<Integer> less\_100 = x -> (x < 100);

System.out.println(grt\_10.and(less\_100).test(60));

**or(Predicate P):**

Predicate<Integer> eq\_10 = x -> (x == 10);

Predicate<Integer> grt\_20 = x -> (x > 20);

System.out.println(eq\_10.or(grt\_20).test(21));

**negate():**

Predicate<Integer> greater\_than = x -> (x > 10);

System.out.println(greater\_than.negate().test(11));

**Useful Tips:**

When we want to print or show stream data, we have to collect it into collections by using **collect()** method. Ex..

// To print we have to use collect() method

List<Integer> data = filteredData.collect(Collectors.toList());

System.out.println(data);

Or

List<Integer> data = list.stream().filter(predicate).collect(Collectors.toList());

System.out.println(data);

**Useful Tips:**

Once we change Collection into stream, we can’t reuse the steam. Ex..

Stream<Integer> **nStream** = **Stream**.*of*(123, 234, 11, 57, 60, -4);

List<Integer> **evenN** = **nStream**.**filter**(integer -> integer % 2 == 0).**collect**(Collectors.*toList*());  
  
 List<Integer> **oddN** = **nStream**.**filter**(integer -> integer % 2 == 1).**collect**(Collectors.*toList*());

Exception in thread "main" java.lang.IllegalStateException: **stream has already been**

**operated upon or closed**

**Solution:**

One solution for this problem would be creating a new stream every time before every stream

processing, therefore we use the ***Supplier.get()*** method of the Supplier Functional Interface.

**Supplier**<Stream<Integer>**>** **nStream** = **() ->** **Stream**.*of*(123, 234, 11, 57, 60, -4);

List<Integer> **evenN** = **nStream.get()**.**filter**(integer -> integer % 2 == 0).**collect**(Collectors.*toList*());  
  
 List<Integer> **oddN** = **nStream.get().filter**(integer -> integer % 2 == 1).**collect**(Collectors.*toList*());

**Useful Tips:**

But we can use same Collection to change it into stream.

List<Integer> **nStream** = **Stream**.*of*(123, 234, 11, 57, 60, -4);

List<Integer> **evenN** = **nStream**.**filter**(integer -> integer % 2 == 0).**collect**(Collectors.*toList*());  
  
 List<Integer> **oddN** = **nStream**.**filter**(integer -> integer % 2 == 1).**collect**(Collectors.*toList*());

**Useful Tips:**

Convert **List String** to **List Integer** or **Double** or **Long** and **Vis versa**:

**List**<Integer> intR = intNumberString.**stream**().**map**(**Integer::*valueOf***).**collect**(Collectors.*toList*());  
Vv  
**List**<String> strR = integerList.**stream**().**map**(**String::*valueOf***).**collect**(Collectors.*toList*()); // or  
**List**<String> r = integerList.stream().map(Object::toString).collect(Collectors.*toList*());  
  
Convert **List String** to **List Char** Vis versa:   
**List**<Character> chR = stringList.**stream**().**flatMapToInt**(**x -> x.chars()**).**mapToObj**(**i -> (char**) .**collect**(Collectors.*toList*());  
Vv  
**String** stR = chR.**stream**().**map( u-> u.toString()**).**collect**(**Collectors.*joining*()**);

Convert **List** to **Array** Vis versa:  
**int**[] intArray = integerList.**stream**().**mapToInt**(**i->i**).toArray();  
**String**[] strArray = stringList.**stream**().**toArray**(**n -> new String[stringList.size()]**);  
Vv  
**List**<String> arrayToList = Arrays.***asList***(strArray);  
**List**<Integer> IntList = Arrays.***stream***(intArray).**boxed**().**collect**(Collectors.*toList*());

**Useful Tips:**

***groupingBy():*** method returns a *Collector* implementing a “*GROUP BY*” operation on *Stream* elements and

returns the result as a *Map*.

**Syntax**

**groupingBy**(classifier)

**groupingBy**(classifier, collector)

**groupingBy**(classifier, supplier, collector)

**We can pass the following arguments to this method:**

* + **classifier:**

maps input elements to map keys

* + **collector:**

is the downstream reduction function. By default, *Collectors.toList()* is used which causes

the grouped elements into a *List*.

* + **supplier:**

provides a new empty Map into which the results will be inserted. By default,

***HashMap::new*** is used. We can use other maps such as **TreeMap, LinkedHashMap or**

**ConcurrentMap** to insert additional behavior in the grouping process such as sorting.

// Get all Employees' Grouped by city and save it into List

**Map**< Object, **List**<**Employee**>> groupByCity = listOfEmployee.**stream**()  
 .**collect**(Collectors.***groupingBy***(result -> result.getAddresses().getCity()));  
 groupByCity.forEach((k,v)-> System.*out*.println(" City =>>> " + k + " | Employee =>>> " + v));  
  
// Get all Employees' Grouped by state and save it into Set  
**Map**< Object, **Set**<**Employee**>> groupByState = listOfEmployee.**stream**()  
 .**collect**(Collectors.***groupingBy***(result -> result.getAddresses().getState(), TreeMap::new,

Collectors.*toSet*()));

**Useful Tips:**

Only we can apply stream() on a collection while using Intermediate or Terminal operation.

**Java 8 Stream API Cheat Sheet**

**Graphical user interface

Description automatically generated**

**Intermediate operations:**

* + Intermediate operations are always **lazy** which means they will not be executed till a Terminal operation called.
  + An intermediate operation returns a new stream, they can be chained. **Stream<T> distinct().**

**filter(Predicate<T**>):

* + It filters each element with the given predicate and return filtered stream.

List<String> s = sList.stream().**filter**(x -> !x.startsWith("A")).collect(Collectors.toList());

**map(Function<T>):**

* + Converts each element in the stream into another object. It can also ransform object to anoterh type.

List<String> s = sList.stream().**map**(s -> s.toLowerCase()).collect(Collectors.*toList*());

**flatMap(Function<T>):**

* + Flatten a stream of colletction into stream of object. It merges multiple collections/arrays into one.

List<String> s = sList.stream().**flatMap**(s -> s.stream()).collect(Collectors.toList());

or

List<String> s = Stream.*of*(stram1, stream2).flatMap(x -> x.stream()).collect(Collectors.toList());

**flatMapToInt() , flatMapToDuble, flatMapToLong.**

**sorted(Comparator<T>):**

* + Returns a sorted stream in the natural order by default (stream.sorted()). we can provide our own comparator for the desired sorting order.

List<String> s = sList.stream().**sorted**((s, ss) -> s.compareTo(ss)).collect(Collectors.toList());

**peek(Consumer<T>):**

* + performing the provided action on each element as elements are consumed from the resulting stream.

sList.stream().distinct().filter(s -> s.length() < 10)

.**peek**(s -> System.out.println(" Peek Filtered value: :=>" + s))

.map(String::toUpperCase)

.**peek**(s -> System.out.println("Peek Mapped value: :=>" + s))

.collect(Collectors.toList());

**distinct():**

* + Returns a stream consisting of the distinct elements of this stream.

List<String> s = sList.stream().**distinct**().collect(Collectors.toList());

**limit(long n):**

* + Returns a stream with elements of specified number.

List<String> s = sList.stream().**limit**(3).collect(Collectors.toList());

**skip(long n):**

* + returns a new stream consisting of the remaining elements of the original Stream

List<String> s = sList.stream().**skip**(3).collect(Collectors.toList());

**parallel():**

* + It executes code parallelly by dividing single core into multiple core. We can create it by using **parallel()** or **parallelStream()** method on collection.

Long result = list.parallelStream().filter(s -> s.contains(“j”)).count());

System.out.println(“result: “ + result); or

IntStream intSt = IntStream.*rangeClosed*(1,10);  
 System.*out*.println(" Result: " + intSt.**parallel()**.filter(s -> s > 4).count());

**Terminal operations:**

* + Terminal operation traverse the stream and execute the pipeline of intermediate operations to produce the result. Terminal operations are not lazy i.e. eager.
  + A stream can have only one terminal operation, it cannot be chained.
  + It produces either a value of a side-effect **Optional<T> findAny()**

**forEach():**

* + It iterates through each element and perform action. Doesn’t guarantee the order of elements.

sList.stream().**forEach**((s) -> System.*out*.println("forEach() :=> " + s)); or

sList.stream().forEach(System.out::println);

**forEachOrdered():**

* + It iterates through each element and perform action. Order of elements are guaranteed. sList.stream().**forEachOrdered**((s) -> System.*out*.println("forEachOrdered() :=> " + s)); or

sList.stream().**forEachOrdered** (System.out::println);

**collect():**

* + It used to receive elements from a stream and store them in a collection.

List<String> s = sList.stream().map((s) -> s.toUpperCase()).**collect**(Collectors.*toList*());

**findFirst():**

It returns an **Optional** describing the **first element of this stream**

Optional<String> s = sList.stream().**findFirst**();

**findAny():**

* + Returns an **Optional** describing the any element of this stream. Order of elements are not guaranteed.

Optional<String> s = sList.stream().**findAny**();

**anyMatch(Predicate<T**>**):**

It is a **short**-**circuiting operation**. It is used to check if the Stream contains at least one element that satisfies the given perdicate.

boolean s = sList.stream().**anyMatch**((s) -> s.startsWith("A"));

**allMatch(Predicate<T**>**):**

* + It is a **short**-**circuiting operation**. It Returns true or false if the predicate matches all elements.

boolean s = sList.stream().**allMatch**((s) -> s.contains("Juice"));

**noneMatch(Predicate<T**>**):**

* + It is a **short**-**circuiting operation**. It Returns true or false if the predicate dosen’t matche any element.

boolean s = sList.stream().**noneMatch**((s) -> s.contains("Potato"));

**min(Comparator<T>):**

* + It returns an **Optional** describing the minimum element of the stream.

Optional<String> sMin = sList.stream().**min**((x, y) -> x.compareTo(y));

**max(Comparator<T>):**

* + It returns an **Optional** describing the maximum element of the stream.

Optional<String> sMax = sList.stream().**max**((x, y) -> x.compareTo(y)); or

Optional<String> sMax = sList.stream().**max(**Comparator.comparing(name));

**count():**

* + It counts the filtered result and return counted numbers.

long countString = stringList.stream().filter(x -> x.startsWith("A")).**count**();

**toArray():**

* + It returns an array containing the elements of the given stream.

Object[] s = sList.stream().**toArray**(); or

String[] s = stringList.toArray(s -> new String[s]);

**reduce():**

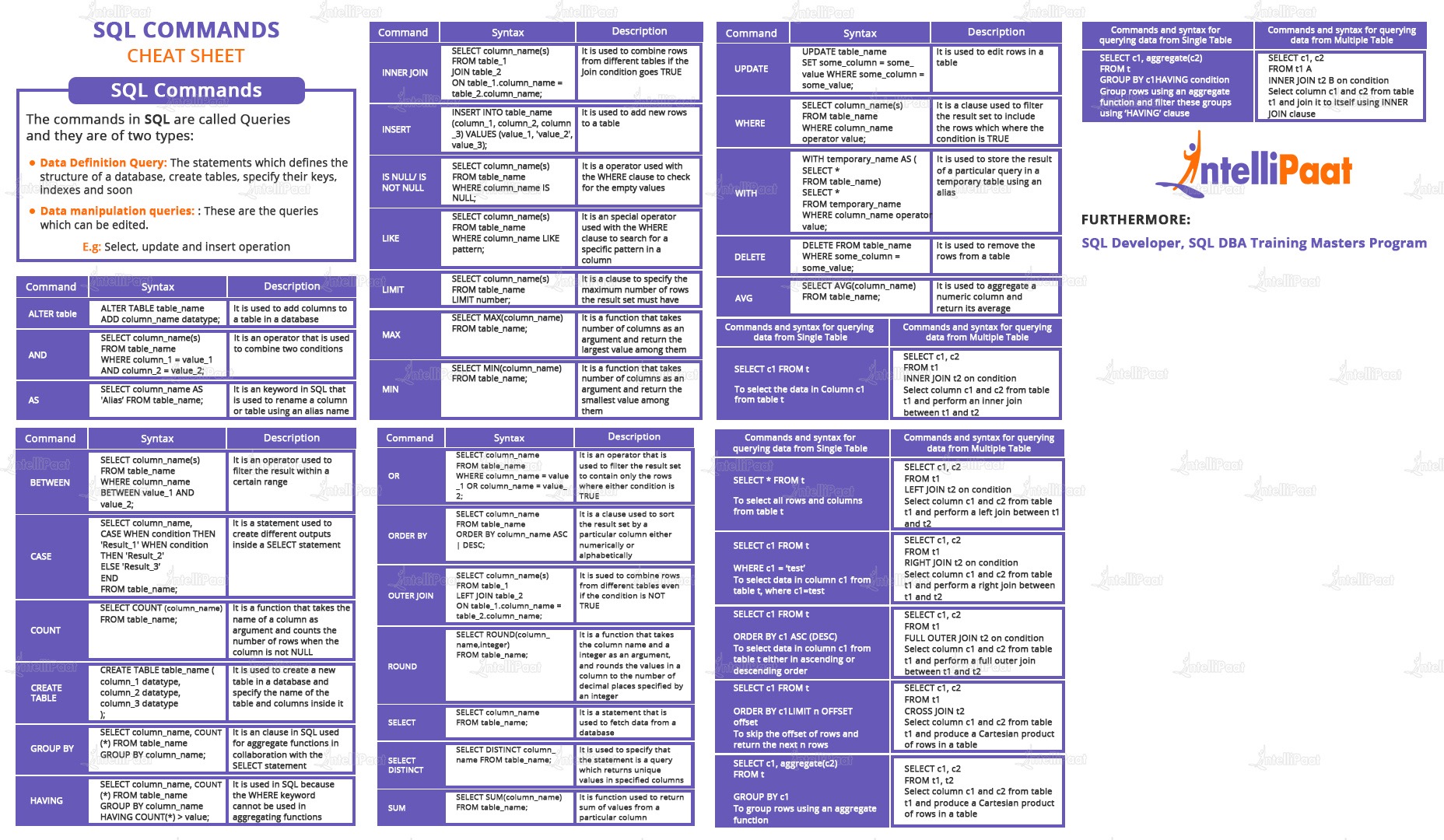
* + It performs a reduction on the elements of the stream with the given function.
  + **T reduce(T identity, BinaryOperator<T> accumulator);**

**int reduce(int identity, IntBinaryOperator op);**

* + - **identity** = default or initial value.
    - **BinaryOperator** = functional interface, take two values and produces a new value.
  + If the identity argument is missing, there is no default or initial value, and it returns an **Optional**.
    - **Optional<T> reduce(BinaryOperator<T> accumulator);**

String s = sList.stream().**reduce**("Type of Food: ", (x, y) -> x + “ | “ + y);

**SQL**

****

**DBMS:**

A database management system or [DBMS](https://intellipaat.com/blog/what-is-dbms/) is system software that can create, retrieve, update, and manage a

database.

**Relational Database:**

A database relationship in the form of tables. The tables are related to each other based on data common to

each.

**Entity:**

A representation of a real object that is uniquely identifiable and contains attributes. An entity resides in a table, it is a single set of information.

**ACID  (Atomicity, Consistency, Isolation, Durability):**

- **ACID** is especially concerned with how a database recovers from any failure that might occur while processing a transaction.

- An ACID-compliant DBMS ensures that the data in the database remains accurate and consistent despite any such failures.

**Atomicity:**

**-** This implies that if any aspect of a transaction fails, the whole transaction fails and the database state remains unchanged.

**Consistency:**

**-** The transaction never leaves the database without finishing its state.

**Isolation:**

**-** No transaction will be affected by any other transaction. So a transaction cannot read data

from any other transaction that has not yet completed.

**Durability:**

**-** Once a transaction is committed, it will occur regardless of what happens in between such as a power outage, fire, or some other kind of disturbance.

**Table:**

A table is a group of fields with certain parameters. A structure that organizes data into rows and columns –

forming a grid

**SQL:**

Is a standard language for storing, manipulating and retrieving data in databases. SQL stands for Structured

Query Language.

**Diagram

Description automatically generated**

**The main subsets of SQL are:**

Data Definition Language (DDL):

-  Is that part of SQL that defines the data structure of the database in the initial stage when the

database is about to be created.

Data Manipulation Language (DML):

- Is used to manipulate already existing data in a database

Data Control Language (DCL):

- Is used to control access to the data in the database. DCL commands are normally used to create

objects related to user access and to control the distribution of privileges among users.

Transaction Control Language (TCL):

**-** TCL is used to control the changes made by DML commands.

**RF:**

- <https://www.w3schools.com/sql/default.asp>

**Useful Tips:**

**Null value**: It is a field left with blank during record creation. It is different from a zero value or a field that

contains spaces. It is not possible to test for NULL values with comparison operators, such as =,

<, or <>. We will have to use the IS NULL and IS NOT NULL operators instead.

SELECT column\_namesFROM table\_name  
WHERE column\_name IS NULL;

**SQL Order of Execution**

- Each part of the query is executed sequentially, so it’s important to understand the order of execution:

1. **FROM** and **JOIN**:  First executed to determine the total working set of data that is being queried
2. **WHERE**: These constraints are applied to the individual rows.
3. **GROUP BY**:  Grouped based on common values in the column specified in the GROUP BY clause.
4. **HAVING**: If the query has a **GROUP BY** clause, then the constraints in the **HAVING** clause are
5. applied to the grouped rows.
6. **SELECT**: Any expressions in the **SELECT** part of the query are finally computed.
7. **DISTINCT**: Rows with duplicate values in the column marked as **DISTINCT** will be discarded.
8. **ORDER BY**:  Rows are then sorted by the specified data in either ascending or descending order.
9. **LIMIT**: Finally, the rows that fall outside the range specified by the LIMIT are discarded.

**Most Used SQL Command**

**SELECT:**

* + Extracts data from a database
  + To select specific fields from database we use

SELECT column1, column2, ...  
 FROM table\_name;

Or SELECT \* FROM table\_name;

**SELECT** **DISTINCT:**

* + It selects non repeated fields

SELECT DISTINCT column1, column2, ...  
FROM table\_name;

**WHERE:**

* + Is used to filter records. It is used to extract only those records that fulfill a specified condition.

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

* + Ex to get all list of records whose department

SELECT \* FROM **Employe**  
WHERE dno=3;

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Employee Table** | | | | | | |
| **ssn** | **ename** | **bdate** | **address** | **sex** | **salary** | **dno** |
| 1011 | Ermi | 1990-10-10 | Main St | M | 100.5 | 1 |
| 1012 | Jim | 1991-10-10 | Long St | M | 100.6 | 1 |
| 1013 | Loul | 1992-10-10 | Marconi St | M | 100.8 | 3 |
| 1014 | Tomas | 1993-10-10 | Jeferson St | M | 100.9 | 2 |
| 1015 | Saral | 1994-10-10 | Hampshare St | F | 200 | 2 |
| 1016 | Abigael | 1992-10-10 | Marconi St | M | 100.8 | 3 |

Result

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Employe Table** | | | | | | |
| **ssn** | **ename** | **bdate** | **address** | **sex** | **salary** | **dno** |
| 1013 | Loul | 1992-10-10 | Marconi St | M | 100.8 | 3 |
| 1016 | Abigael | 1992-10-10 | Marconi St | M | 100.8 | 3 |

* + The WHERE clause can be combined with AND, OR, and NOT operators.

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 AND condition2 AND condition3 ...;

SELECT column1, column2, ...  
FROM table\_name  
WHERE condition1 OR condition2 OR condition3 ...;

**ORDER BY:**

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

SELECT column1, column2, ...  
FROM table\_name  
ORDER BY column1, column2, ... ASC|DESC;

**UPDATE:**

* + Is used to modify the existing records in a table. If we don’t use WHERE statement WITH UPDATE, the change will reflet to all records of the table.

UPDATE table\_name  
SET column1 = value1, column2 = value2, ...  
WHERE condition;

**INSERT** **INTO:**

* + It is used to insert new records in a table.

INSERT INTO table\_name   
VALUES (value1, value2, value3, ...);

* + To insert into specific table column we use bellow syntax

INSERT INTO table\_name (column1, column2, column3, ...)  
VALUES (value1, value2, value3, ...);

**DELETE:**

* + Is used to delete existing records in a table.
  + To delete specific record.

DELETE FROM table\_name WHERE condition;

* + To delete all records.

DELETE FROM table\_name;

**SQL TOP, LIMIT and FETCH FIRST:**

* + Is selects the specified records and return from the table.

For SQL Server/MS Access

SELECT TOP 3 \*

FROM Customers  
WHERE Country='Germany';

Or

For MySQL:

SELECT \*

FROM Customers  
WHERE Country='Germany'  
LIMIT 3;

Or

For Oracle

SELECT \* FROM Customers  
WHERE Country='Germany'  
FETCH FIRST 3 ROWS ONLY;

**AGGREGATION (AVG(), COUNT(), SUM(), MIN(), and MAX()):**

**-** They areis used to perform calculations on multiple values and return the result in a single

value like the average of all values.

- Aggregation may not appear in the **WHERE** clause unless it is in a subquery contained in the **HAVING** clause or a select list. EX…

wrong query

SELECT  gender, AVG (age) FROM  Employee WHERE  AVG (age)>30 GROUP BY gender

**MIN and MAX:**

**MIN:**

**-** Min return the smallest value of the selected column.

SELECT MIN(column\_name)  
FROM table\_name

- As used to change column name price into SmallestPrice.

SELECT MIN(Price) AS SmallestPrice  
FROM Products;

**MAX:**

- Max return the largest value of the selected column.

SELECT MAX(column\_name)  
FROM table\_name  
WHERE condition;

**COUNT(), AVG() and SUM()**

**COUNT():**

- It returns the number of rows that matches a specified criterion. It can be used without

WHERE.

SELECT COUNT(column\_name)  
FROM table\_name  
WHERE condition;

**AVG():**

- It returns the average value of a numeric column. It can be used without WHERE.

SELECT AVG(column\_name)  
FROM table\_name  
WHERE condition;

**SUM():**

**-** It returns the total sum of a numeric column. It can be used without WHERE.

SELECT SUM(column\_name)  
FROM table\_name  
WHERE condition;

**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

SELECT column1, column2, ...  
FROM table\_name  
WHERE columnN LIKE pattern;

**Like Operators with ‘%’ and ‘\_’ wildcards:**

|  |  |
| --- | --- |
| **LIKE Operator** | Description |
| WHERE CustomerName LIKE 'a%' | **Finds any values that start with "a"** |
| WHERE CustomerName LIKE '%a' | **Finds any values that end with "a"** |
| WHERE CustomerName LIKE '%or%' | **Finds any values that have "or" in any position** |
| WHERE CustomerName LIKE '\_r%' | **Finds any values that have "r" in the second position** |
| WHERE CustomerName LIKE 'a\_%' | **Finds any values that start with "a" and are at least 2 characters in length** |
| WHERE CustomerName LIKE 'a\_\_%' | **Finds any values that start with "a" and are at least 3 characters in length** |
| WHERE ContactName LIKE 'a%o' | **Finds any values that start with "a" and ends with "o"** |

* + Selects all customers with a CustomerName starting with "a":

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a%';

* + Selects all customers with a CustomerName ending with "a":

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%a';

* + Selects all customers with a CustomerName that have "or" in any position:

SELECT \* FROM Customers  
WHERE CustomerName LIKE '%or%';

* + Selects all customers with a CustomerName that have "r" in the second position:

SELECT \* FROM Customers  
WHERE CustomerName LIKE '\_r%';

* + Selects all customers with a CustomerName that starts with "a" and are at least 3 characters in length:

SELECT \* FROM Customers  
WHERE CustomerName LIKE 'a\_\_%';

* + Selects all customers with a ContactName that starts with "a" and ends with "o":

SELECT \* FROM Customers  
WHERE ContactName LIKE 'a%o';

* + Selects all customers with a CustomerName that does NOT start with "a":

SELECT \* FROM Customers  
WHERE CustomerName NOT LIKE 'a%';

**Wildcard Characters in SQL Server:**

|  |  |  |
| --- | --- | --- |
| Symbol | Description | Example |
| % | Represents zero or more characters | bl% finds bl, black, blue, and blob |
| \_ | Represents a single character | h\_t finds hot, hat, and hit |
| [] | Represents any single character within the brackets | h[oa]t finds hot and hat, but not hit |
| ^ | Represents any character not in the brackets | h[^oa]t finds hit, but not hot and hat |
| - | Represents any single character within specified range | c[a-b]t finds cat and cbt |

* + Selects all customers with a City starting with "b", "s", or "p":

SELECT \* FROM Customers  
WHERE City LIKE '[bsp]%';

- Selects all customers with a City starting with "a", "b", or "c":

SELECT \* FROM Customers  
 WHERE City LIKE '[a-c]%';

- Select all customers with a City NOT starting with "b", "s", or "p":

SELECT \* FROM Customers  
 WHERE City LIKE '[!bsp]%';

**AS (Alias):**

- Aliases are used to give a table, or a column in a table, a temporary name.

- An alias only exists for the duration of that query.

SELECT column\_name AS alias\_name  
FROM table\_name;

- An alias is represented explicitly by the **AS** keyword.

SELECT  A.emp\_name **AS** "Employee" /\* Alias using AS keyword \*/

B.emp\_name **AS** "Supervisor"

FROM  employee A, employee B /\* Alias without AS keyword \*/

WHERE  A.emp\_sup = B.emp\_id;

**CONCAT:**

- Adds two or more strings together

CONCAT(string1, string2, ...., string\_n)

SELECT CONCAT('SQL', ' is', ' fun!');

**List the different types of relationships in SQL:**

* **One-to-One** - Relationship between two tables where each record in one table is associated with the maximum of one record in the other table.
* **One-to-Many** & **Many-to-One** - This is the most commonly used relationship where a record in a table is associated with multiple records in the other table.
* **Many-to-Many** - This is used in cases when multiple instances on both sides are needed for

defining a relationship.

* **Self-Referencing Relationships** - This is used when a table needs to define a relationship with itself.

**JOIN:**

-  Is used to combine rows from two or more tables, based on a related column between them.

**Here are the different types of the JOINs in SQL:**

**INNER JOIN:**

- Returns records that have matching values in both tables.



SELECT column\_name(s)   
FROM table1   
INNER JOIN table2 ON table1.column\_name = table2.column\_name;

**LEFT OUTER JOIN:**

- Returns all records from the left table, and the matched records from the right table.



SELECT column\_name(s)  
FROM table1  
LEFT JOIN table2ON table1.column\_name = table2.column\_name;

**RIGHT OUTER JOIN:**

- Returns all records from the right table, and the matched records from the left table



SELECT column\_name(s)  
FROM table1  
RIGHT JOIN table2ON table1.column\_name = table2.column\_name;

**FULL OUTER JOIN:**

- Returns all records when there is a match in either left or right table



SELECT column\_name(s)  
FROM table1  
FULL OUTER JOIN table2ON table1.column\_name = table2.column\_nameWHERE condition;

**SELF JOIN:**

  - A self-join is a regular join, but the table is joined with itself.

SELECT column\_name(s)  
FROM table1 T1, table1 T2  
WHERE condition;

T1 and T2 are different table aliases for the same table.

**UNION:**

- It is used to combine the result-set of two or more SELECT statements.

- Every **SELECT** statement within **UNION** must have the same number of columns.

- The columns must also have similar data types.

- The columns in every SELECT statement must also be in the same order.

- UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL.

SELECT column\_name(s) FROM table1  
 UNION  
 SELECT column\_name(s) FROM table2;

**GROUP BY:**

- It is used to groups rows that have the same values into summary rows.

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)

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

- Lists the number of customers in each country:

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country;

- Whenever we are working with aggregate functions and are using the **GROUP BY** clause, we use **HAVING** clause instead of **WHERE** because WHER used for filtering data before aggregation.

- Difference between HAVING and WHERE is HAVING filter data after aggregations.

- List the number of customers in each country, ordered by the country with the most customers first.

SELECT COUNT(CustomerID), Country

FROM  Customers

GROUP BY Country;

ORDER BY COUNT(CustomerID) DESC;

**HAVING:**

- The **HAVING** clause was added to SQL because the **WHERE** keyword cannot be used with

aggregate functions.

SELECT column\_name(s)  
FROM table\_name  
WHERE condition  
GROUP BY column\_name(s)HAVING conditionORDER BY column\_name(s);

- The following SQL statement lists the number of customers in each country. Only include countries

with more than 5 customers:

SELECT COUNT(CustomerID), Country  
FROM Customers  
GROUP BY Country  
HAVING COUNT(CustomerID) > 5;

**SQL Logical Operators:**

- The Logical operators are those that are true or false.

|  |  |
| --- | --- |
| **Operator** | Description |
| ALL | **TRUE if all of the subquery values meet the condition** |
| AND | **TRUE if all the conditions separated by AND is TRUE** |
| ANY | **TRUE if any of the subquery values meet the condition** |
| BETWEEN | **TRUE if the operand is within the range of comparisons** |
| EXISTS | **TRUE if the subquery returns one or more records** |
| IN | **TRUE if the operand is equal to one of a list of expressions** |
| LIKE | **TRUE if the operand matches a pattern** |
| NOT | **Displays a record if the condition(s) is NOT TRUE** |
| OR | **TRUE if any of the conditions separated by OR is TRUE** |
| SOME | **TRUE if any of the subquery values meet the condition** |

**AND:**

- AND Operator allows you to connect multiple statement

SELECT \*

FROM CUSTOMERS

WHERE AGE >= 25 AND SALARY >= 6500;

**OR:**

- OR Operator allows you to connect multiple statement

SELECT \*

FROM CUSTOMERS

WHERE AGE >= 25 OR SALARY >= 6500;

**IN:**

- IN Operator allows you to specify multiple values in a WHERE clause.

SELECT column\_name(s)  
 FROM table\_name  
 WHERE column\_name IN (value1, value2, ...);

OR

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name IN (*SELECT* STATEMENT);

- Statement selects all customers that are located in "Germany", "France" or "UK":

SELECT \* FROM Customers  
 WHERE Country IN ('Germany', 'France', 'UK');

- Selects all customers that are from the same countries as the suppliers:

SELECT \* FROM Customers  
 WHERE Country IN (SELECT Country FROM Suppliers);

**BETWEEEN:**

- This operator selects values within a given range.

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name BETWEEN value1 AND value2;

**EXISTS:**

- Is used to test for the existence of any record in a subquery. The **EXISTS** operator returns TRUE if

the subquery returns one or more records.

SELECT column\_name(s)  
FROM table\_name  
WHERE EXISTS  
(SELECT column\_name FROM table\_name WHERE condition);

- The following SQL statement returns TRUE and lists the suppliers with a product price less than 20.

SELECT SupplierName  
FROM Suppliers  
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID = Suppliers.supplierID AND Price < 20);

**ANY:**

- The condition will be true if the operation is true for any of the values in the range.

SELECT column\_name(s)  
FROM table\_name  
WHERE column\_name operator ANY  
  (SELECT column\_name  FROM table\_name  WHERE condition);

**ALL:**

- The condition will be true only if the operation is true for all values in the range.

SELECT ALL column\_name(s)  
FROM table\_name  
WHERE condition;

- Lists the ProductName if it finds ANY records in the OrderDetails table has Quantity larger than 99

(this will return TRUE because the Quantity column has some values larger than 99):

SELECT ProductName  
FROM Products  
WHERE ProductID = ANY  
  (SELECT ProductID  
  FROM OrderDetails  
  WHERE Quantity > 99);

**NULL:**

- NULL Operator allows you to connect multiple statement

SELECT \*

FROM CUSTOMERS

WHERE AGE IS NOT NULL;

**SELECT INTO:**

- It copies data from one table into a new table.

SELECT \*  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

- Copy only some columns into a new table:

SELECT column1, column2, column3, ...  
INTO newtable [IN externaldb]  
FROM oldtableWHERE condition;

- This statement creates a backup copy of Customers:

SELECT \* INTO CustomersBackup2017  
 FROM Customers;

- This statement uses the **IN** clause to copy the table into a new table in another database:

SELECT \* INTO CustomersBackup2017 IN 'Backup.mdb'  
FROM Customers;

- The following SQL statement copies only the German customers into a new table:

SELECT \* INTO CustomersGermany  
FROM Customers  
WHERE Country = 'Germany';

**INSERT INTO SELECT:**

- It copies data from one table and inserts it into another table.

- Statement requires that the data types in source and target tables match.

INSERT INTO table2  
SELECT \* FROM table1WHERE condition;

- Copy only some columns from one table into another table:

INSERT INTO table2 (column1, column2, column3, ...)  
SELECT column1, column2, column3, ...  
FROM table1  
WHERE condition;

- Statement copies "Suppliers" into "Customers" (the columns with no data, will contain NULL):

INSERT INTO Customers (CustomerName, City, Country)  
SELECT SupplierName, City, Country FROM Suppliers;

**SQL Stored Procedures:**

- It is a saved query that can be called with parameter or without parameter.

CREATE PROCEDURE procedure\_name  
AS  
sql\_statement  
GO;

Calling the procedures.

EXEC procedure\_name;

- Stored Procedure with one parameter

CREATE PROCEDURE SelectAllCustomers @City nvarchar(30)  
AS  
SELECT \* FROM Customers WHERE City = @City  
GO;

Calling the procedures.

EXEC SelectAllCustomers @City = 'London';

**CREATE DATABASE:**

- It used to create a new SQL database.

CREATE DATABASE databasename;

**DROP DATABASE:**

- It used to drop an existing SQL database.

DROP DATABASE databasename;

**BACKUP DATABASE:**

- It used in SQL Server to create a full back up of an existing SQL database

BACKUP DATABASE databasename  
TO DISK = 'filepath';

- A differential back up only backs up data that have changed since the last full database backup.

BACKUP DATABASE databasename  
TO DISK = 'filepath'  
WITH DIFFERENTIAL;

- The statement creates a full back up of the existing database "testDB" to the D disk:

BACKUP DATABASE testDB  
TO DISK = 'D:\backups\testDB.bak';

**CREATE TABLE:**

- It used to create a new table in a database.

CREATE TABLE table\_name (  
    column1 datatype,  
    column2 datatype,  
    column3 datatype,  
   ....  
);

**DROP TABLE:**

- Drops an existing table in a database.

DROP TABLE table\_name;

**TRUNCATE TABLE:**

- Deletes the data inside a table, but not the table itself.

TRUNCATE TABLE Categories;

**ALTER TABLE:**

- It used to add, delete, or modify columns in an existing table. Also used to add and drop various

constraints on an existing table.

ALTER TABLE table\_name  
ADD column\_name datatype;

-  Deletes the "Email" column from the "Customers" table:

ALTER TABLE Customers  
DROP COLUMN Email;

**PRIMARY KEY:**

- Primary Key constraint uniquely identifies each record in a table.

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    PRIMARY KEY (ID)  
);

**FOREIGN KEY:**

- It is a field (or collection of fields) in one table, that refers to the **PRIMARY KEY** in another table.

- The table with the foreign key is called the child table, and the table with the primary key is called

parent table.

- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key

column, because it has to be one of the values contained in the parent table.

- "PersonID" column in the "**Orders**" table points to the "PersonID" column in the "**Persons**" table.

- "PersonID" column in the "**Persons**" table is the PRIMARY KEY in the "**Persons**" table.

- "PersonID" column in the "**Orders**" table is a FOREIGN KEY in the "**Orders**" table.

Persons Table

|  |  |  |  |
| --- | --- | --- | --- |
| PersonID | LastName | FirstName | Age |
| 1 | Hansen | Ola | 30 |
| 2 | Svendson | Tove | 23 |
| 3 | Pettersen | Kari | 20 |

Orders Table

|  |  |  |
| --- | --- | --- |
| OrderID | OrderNumber | PersonID |
| 1 | 77895 | 3 |
| 2 | 44678 | 3 |
| 3 | 22456 | 2 |
| 4 | 24562 | 1 |

- Creates a FOREIGN KEY on the "**PersonID**" column when the "**Orders**" table is created:

CREATE TABLE Orders (  
    OrderID int NOT NULL,  
    OrderNumber int NOT NULL,  
    PersonID int,  
    PRIMARY KEY (OrderID),  
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)  
);

**Difference Between Primary Key and Unique Key:**

- Both primary and unique keys carry unique values but a primary key cannot have a null value, while a unique key can. In a table, there cannot be more than one primary key, but there can be multiple unique keys.

**AUTO-INCREMENT:**

- Automatically generate a unique number whenever a new record is inserted into a table.

CREATE TABLE Persons (  
    ID int NOT NULL AUTO-INCREMENT,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
);

**CHECK:**

- Constraint is used to limit the value range that can be placed in a column.

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    CHECK (Age>=18)  
);

**DEFAULT:**

- Constraint is used to set a default value for a column.

CREATE TABLE Persons (  
    ID int NOT NULL,  
    LastName varchar(255) NOT NULL,  
    FirstName varchar(255),  
    Age int,  
    City varchar(255) DEFAULT 'Sandnes'  
);

**TRIGGER:**

- Is a stored program invoked automatically in response to an event such as [insert](https://www.mysqltutorial.org/mysql-insert-statement.aspx), [update](https://www.mysqltutorial.org/mysql-update-data.aspx), or delete that occurs in the associated table.

CREATE TRIGGER trigger\_name

{**BEFORE** | **AFTER**} {**INSERT** | **UPDATE**| **DELETE** }

ON table\_name FOR EACH ROW

trigger\_body;

- It will be executed before every INSERT statement for the Employee table:

CREATE TRIGGER EmpTrigger

**BEFORE** **INSERT**

ON Employee FOR EACH ROW

IF **NEW**.age < 0

THEN **SET** **NEW**.age = 0; END IF;

**Clustered Index:**

- It is used to sort the rows of data by their key values.

- Table data can be sorted in only way, therefore, there can be only one clustered index per table.

- In SQL Server, the primary key constraint automatically creates a clustered index on that particular

column.

**Non Clustered Index:**

- A non-clustered index doesn’t sort the physical data inside the table. In fact, a non-clustered index is stored at one place and table data is stored in another place.

- This is similar to a textbook where the book content is located in one place and the index is located in another. This allows for more than one non-clustered index per table.

**CREATE I NDEX:**

- Statement is used to create indexes in tables. Indexes are used to retrieve data from the database

more quickly than otherwise. They are just used to speed up searches/queries.

CREATE UNIQUE INDEX index\_name  
ON table\_name (column1, column2, ...);

- Creates an index named "**idx\_lastname**" on the "LastName" column in the "**Persons**" table:

CREATE INDEX idx\_lastname  
ON Persons (LastName);

- Creates an index named “**idx\_pname”** on multiple columns.

CREATE INDEX idx\_pname  
ON Persons (LastName, FirstName);

**DROP INDEX:**

- Statement is used to delete an index in a table.

DROP INDEX index\_name ON table\_name;

OR

ALTER TABLE table\_nameDROP INDEX index\_name;

**CREATE VIEW:**

- is a virtual table based on the result-set of an SQL statement.

- It creates a table with the customized fields.

CREATE VIEW view\_name AS  
SELECT column1, column2, ...  
FROM table\_name  
WHERE condition;

SELECT \* FROM view\_name ;

**SQL Injection:**

- It is a code injection technique that might destroy your database.

- SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

txtUserId = getRequestString("UserId");  
txtSQL = " SELECT \* FROM Users WHERE UserId = " + txtUserId;

- SQL Injection Based on 1=1 is Always True

|  |
| --- |
| 105 OR 1=1 |

User input ==🡺 UserId:

SELECT \* FROM Users WHERE UserId = 105 OR 1=1;

- The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always

TRUE.

**SQL Parameters:**

- To protect a web site from SQL injection, you can use SQL parameters.

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = @0";  
db.Execute(txtSQL,txtUserId);

**SQL HOSTING:**

- Website to store and retrieve data, web server should have access to a database-system that uses the SQL language.

**Most Common SQL hosting databases:**

**- MS SQL Server** (Microsoft's SQL Server)

**- Oracle**

- **MySQL**

- **MS** **Access**

**-** Used for a simple database and it is not well suited for very high-traffic and not powerful as MySQL, SQL Server, or Oracle.

**Normalization:**

- Normalization represents the way of organizing structured data in the database efficiently.

**First Normal Form(1NF):**

- Each column is unique in 1NF.

|  |  |  |
| --- | --- | --- |
| Employee | Age | Department |
| Melvin | 32 | Marketing, Sales |
| Edward | 45 | Quality Assurance |
| Alex | 36 | Human Resource |

|  |  |  |
| --- | --- | --- |
| Employee | Age | Department |
| Melvin | 32 | Marketing |
| Melvin | 32 | Sales |
| Edward | 45 | Quality Assurance |
| Alex | 36 | Human Resource |

**Second Normal Form(2NF):**

- The entity should be considered already in 1NF, and all attributes within the entity should depend solely on the unique identifier of the entity.

Professor Table

|  |  |  |
| --- | --- | --- |
| Professor ID | Professor Name | Course Name |
| P001 | Gene Watson | Intro to Philosophy |
| P002 | Melissa King | Quantum Mechanics |
| P003 | Errol Tyson | Macroeconomics |

Normalized Professor Table

|  |  |
| --- | --- |
| Professor ID | Professor Name |
| P001 | Gene Watson |
| P002 | Melissa King |
| P003 | Errol Tyson |

Normalized Course Table

|  |  |  |
| --- | --- | --- |
| Course ID | Course Name | Professor ID |
| C001 | Intro to Philosophy | P001 |
| C002 | Quantum Mechanics | P002 |
| C003 | Macroeconomics | P003 |

**Third Normal Form(3NF):**

- The entity should be considered already in 2NF, and no column entry should be dependent on

any other entry (value) other than the key for the table.

- If such an entity exists, move it outside into a new table.

- This table isn’t in third normal form because the **Customer zip code** field is dependent on **Customer ID**, which is not this table’s entity key (the entity key here is **Order ID**).

**Order Table**

|  |  |  |  |
| --- | --- | --- | --- |
| Order ID | Order date | Customer ID | Customer zip code |
| R001 | 01/17/2021 | C032 | 99702 |
| R002 | 03/01/2021 | C004 | 39204 |
| R003 | 06/30/2021 | C054 | 06505 |

**Normalized Order Table**

|  |  |  |
| --- | --- | --- |
| Order ID | Order date | Customer ID |
| R001 | 01/17/2021 | C032 |
| R002 | 03/01/2021 | C004 |
| R003 | 06/30/2021 | C054 |

**Normalized Customer Table**

|  |  |
| --- | --- |
| Customer ID | Customer zip code |
| C032 | 99702 |
| C004 | 39204 |
| C054 | 06505 |

**BOYCE-CODD NORMAL FORM(BCNF):**

- 3NF and all tables in the database should be only one primary key.

**Denormalization:**

- Denormalization is the inverse process of normalization, where the normalized schema is converted into a schema that has redundant information.

**SubQuery:**

- Is a query inside another query.  
 - You can only embed one query inside another by following rules :  
 1. subquery can be used in the insert statement.

2. subquery can be used in the select statement as a column.

3. subquery should always return either **scalar value** if used with where clause or **value** from a column if used with **IN**or **NOT** IN clause.

**Noncorrelated Subquery:**

- The inner query doesn't depend on the outer query and can run as a stand-alone query.

- Subquery used along with IN or NOT IN SQL clause.

**SELECT** name

**FROM** City

**WHERE** pincode **IN** (**SELECT** pincode **FROM** pin **WHERE** zone='west')

**Correlated Subquery:**

- Are the one in which **inner query or subquery reference outer query**.  
 **-** Outer query needs to be executed before inner query. One of the most common *example of correlated subquery* is using keywords **exits** and not **exits**

- An important point to note is that **correlated subqueries are slower queries** and one should

avoid it as much as possible.

SELECT \* FROM t1

WHERE column1 = ANY (SELECT column1

FROM t2

WHERE t2.column2 = t1.column2);

**SQL Problems**

################################# Query Problems #######################

RF:

<https://www.edureka.co/blog/interview-questions/sql-query-interview-questions>

**Employee Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| empID | empFname | empLname | department | salary | genger |
| 1 | Jhon | Smith | HR | 5000 | M |
| 2 | Elsa | Ferdinand | Admin | 8000 | F |
| 3 | Ronaldo | Guzman | Account | 3000 | M |
| 4 | Suzan | Alex | HR | 7000 | F |
| 5 | Sajin | Guzman | Security | 10000 | M |
| 6 | Rodrigoz | Garcia | Engineer | 10000 | M |
| 7 | Sashi | Martinez | HR | 5000 | M |
| 8 | Zidan | Petros | Admin | 3500 | M |

**Position Table**

|  |  |  |  |
| --- | --- | --- | --- |
| empID | position | bonas | joiningDate |
| 1 | MANAGER | 1000 | 2001/1/00 |
| 2 | EXCUTIVE | 2000 | 2001/1/10 |
| 3 | LEAD | 1500 | 2002/1/16 |
| 4 | MANAGER | 1000 | 2008/1/01 |
| 5 | SUPER VISER | 150 | 2008/1/08 |
| 6 | TEAM LEAD | 600 | 2002/9/12 |
| 7 | TEAM LEAD | 600 | 2005/6/12 |
| 8 | TEAM LEAD | 600 | 2003/3/12 |
| 8 | TEAM LEAD | 600 | 3200/3/12 |
| 8 | TEAM LEAD | 600 | 2003/3/12 |
| 4 | ROSA GUZMAN | 300 | 2003/3/12 |
| 4 | ROSA GUZMAN | 300 | 2003/3/12 |

Q1, Query empFname in upper Case and use alias empFname as Emplpyee Name.

SELECT UPPER(empFname) AS 'Employee Name'

From Employee;

Q2, Query number of Employee working in HR

SELECT COUNT(empFname)

From Employee

WHERE department = 'HR';

Q3, Query current date:

SELECT CURDATE();

4, Query all Employee whose salary is between 5000 to 10000

SELECT \*

FROM Employee

WHERE salary BETWEEN 5000.00 AND 10000.00;

Q5, Query name of Employee that begin with 'S'

SELECT \*

FROM Employee

WHERE empFname LIKE 'S%';

Q6, Query 2 Employee with salary greater than 7000.00.

SELECT \*

FROM Employee

WHERE salary > 7000.00

LIMIT 2;

Q7, Query All Employees with top salary

SELECT \*

FROM Employee

WHERE Salary = (SELECT MAX(Salary)

FROM Employee);

-Q8, Query Employee with 2 top salary

SELECT \*

FROM Employee

ORDER BY Salary DESC LIMIT 2;

Q9, Query Employee first and last name separated by space as full name?

SELECT CONCAT(empFname, ' ' , empLname) AS 'Full Name'

FROM Employee;

Q10, Query all Employee ordered by last name decendently

SELECT \*

FROM Employee

ORDER BY empLname DESC;

Q11, Query all Employee except Sajin and Rodrigoz?

SELECT \*

FROM Employee

WHERE empFname NOT IN ('Sajin', 'Rodrigoz');

-- Alternating way

SELECT \*

FROM Employee

WHERE empFname != 'Sajin' 'Rodrigoz';

Q12, Write a query to fetch all Employees who also hold the managerial position.

-- Solution using JOIN statement

SELECT \*

FROM Employee e , Position p

WHERE e.empID = p.empID AND p.position = 'MANAGER';

-- Alternating way using INNER JOIN, AND, & IN

SELECT \*

FROM Employee e

INNER JOIN Position p

ON e.empID = p.empID AND p.position IN ('MANAGER');

-- Alternating way using INNER JOIN & WHERE

SELECT \*

FROM Employee e

INNER JOIN Position p

ON e.empID = p.empID

WHERE p.posItion = 'MANAGER';

Q13, Write a query to retrieve employee details from Employee table who have a date of joining in the Position table.

-- Solution using JOIN & EXISTS

SELECT \*

FROM Employee e, Position p

WHERE EXISTS( SELECT \* FROM Position WHERE e.empID = p.empID);

-- Alternating way using JOIN

SELECT \*

FROM Employee e, Position p

WHERE e.empID = p.empID;

Q14, Write a query to retrieve two maximum bonas from the Position table.

-- Solution using colerated query

SELECT DISTINCT bonas FROM Position p

WHERE 2 >= (SELECT COUNT(DISTINCT bonas)

FROM Position p1

WHERE p1.bonas >= p.bonas) ORDER BY p.bonas DESC;

Q15, Write a query to retrieve two maximum bonas from the Position table.

--Solution using colerated query

SELECT DISTINCT bonas FROM Position p

WHERE 2 >= (SELECT COUNT(DISTINCT bonas)

FROM Position p1

WHERE p1.bonas <= p.bonas) ORDER BY p.bonas DESC;

Q16, Write a query to find the 3rd highest salary from the table without using TOP/limit keyword.

SELECT DISTINCT salary FROM Employee e

WHERE 2 = (SELECT COUNT(DISTINCT salary)

FROM Employee e1

WHERE e1.salary > e.salary);

Q17, Write a query to retrieve duplicate records from a table.

SELECT empID, position, COUNT(\*)

FROM Position

GROUP BY empID, position, bonas

HAVING COUNT(\*) > 1;

Q18, Write a query to retrieve the list of employees working in the same department.

SELECT DISTINCT e.empID, e.empFname, e.department

FROM Employee e, Employee e1

WHERE e.department = e1.Department AND e.empID != e1.empID

ORDER BY department;

-- Alternating way using IN

SELECT empID, empFname,department

FROM Employee

WHERE department IN (SELECT department

FROM Employee

GROUP BY department

HAVING COUNT(\*) > 1);

Order Position

With non-duplicate ID

Q19, Write a query to retrieve the last 3 records from the Employee table(No duplicate ID)

(SELECT \*

FROM Position

ORDER BY empID DESC

LIMIT 3)

ORDER BY empID ASC;

--Alternating way using count which return single number

SELECT \*

FROM Employee

WHERE empID > (SELECT COUNT(\*) FROM Employee) - 3;

Q20, Write a query to find the third-highest salary from the Employee table.

-- Common Solution Works for Many Problems

-- Max & Min With dublicate value

-- Solution using Limit & offset. It works for duplicate bonas

-- If the second statment return multiple value, it can't be assigned to boass with WHERE claus

SELECT \*

FROM Position

WHERE bonas = (SELECT DISTINCT bonas

FROM Position

ORDER BY bonas DESC

LIMIT 1 OFFSET 2)

LIMIT 1;

-- Solution only work for non duplicae bonas

SELECT \*

FROM Position

ORDER BY bonas DESC

LIMIT 1 OFFSET 2;

Q21, Write a query to display the first record from the Employee table.

SELECT \*

FROM Employee

LIMIT 1;

-- Alternating way

SELECT \*

FROM Employee

WHERE empID = (SELECT MIN(empID) FROM Employee);

Q22, Write a query to display the last record from the Employee table.

SELECT \*

FROM Employee

ORDER BY empID DESC

LIMIT 1;

-- Alternating way

SELECT \*

FROM Employee

WHERE empID = (SELECT MAX(empID) FROM Employee);

Q23 Write a query to retrieve departments who have less than 2 employees working in it.

SELECT department, COUNT(empID) as 'EmpNo'

FROM Employee

GROUP BY department

HAVING COUNT(empID) = 1;

Q24, Write a query to retrieve Position along with total bonas paid for each of them

SELECT position, SUM(bonas)

FROM Position

GROUP BY position;

Q25, Write a query to fetch 50% records from the Employee table.

SELECT \*

FROM Employee

WHERE empID <= (SELECT COUNT(EmpID)/2

FROM Employee);

**DevOps**

**DevOps:**

Is a combination of software developers (dev) and operations (ops). It is defined as a software engineering methodology which aims to integrate the work of software development and software operations teams by facilitating a culture of collaboration and shared responsibility

**DevOps Lifecycle**

**WaterFall Model**

**Agile Model**