

6) open pom.xml file, then add the below

dependency under <dependencies> tag.

<dependency>

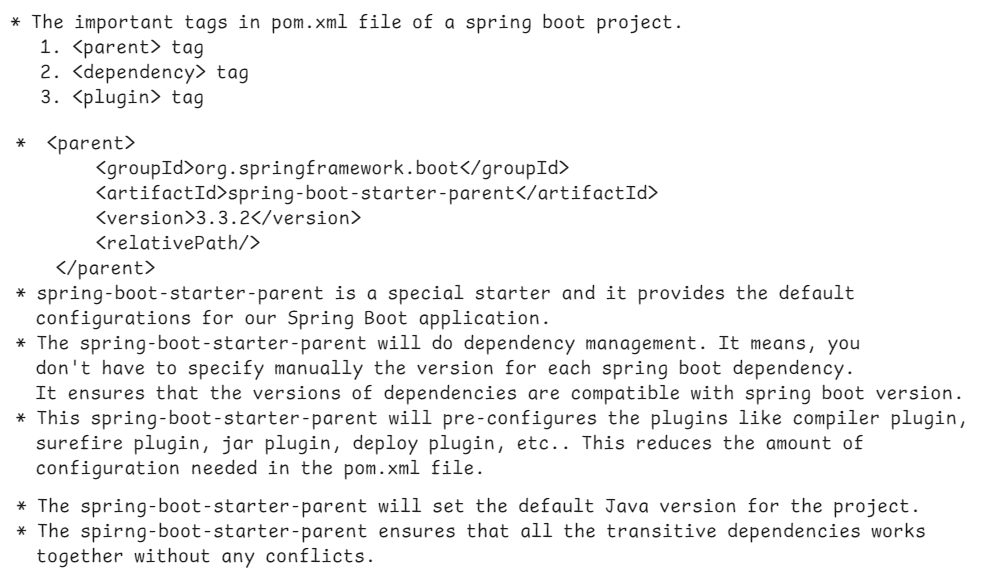
<groupId>org.springframework</groupId>

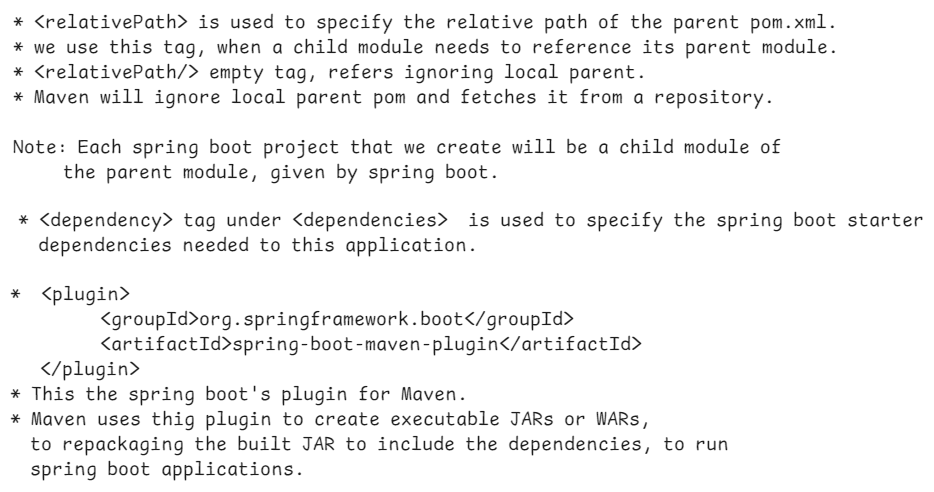
<artifactId>spring-context</artifactId>

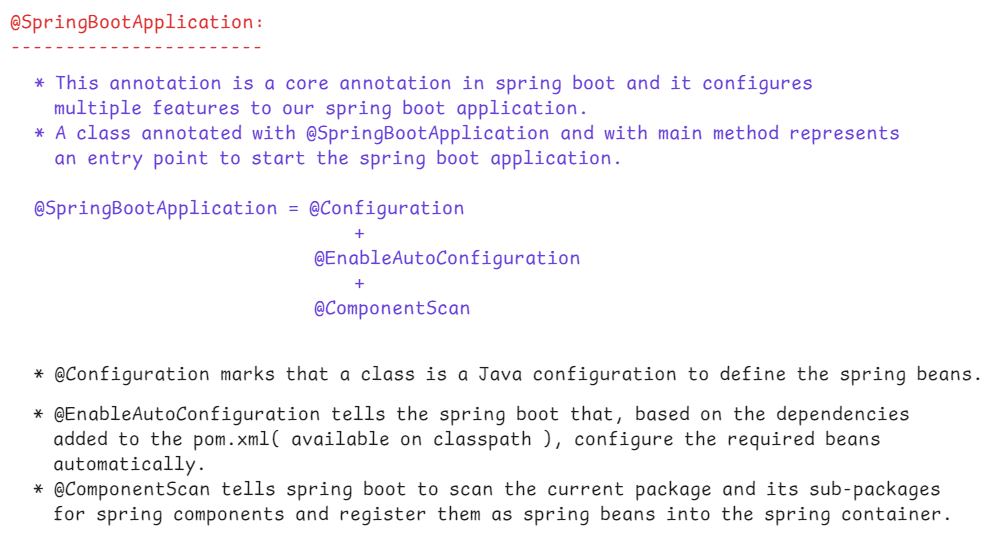
<version>5.3.37</version>

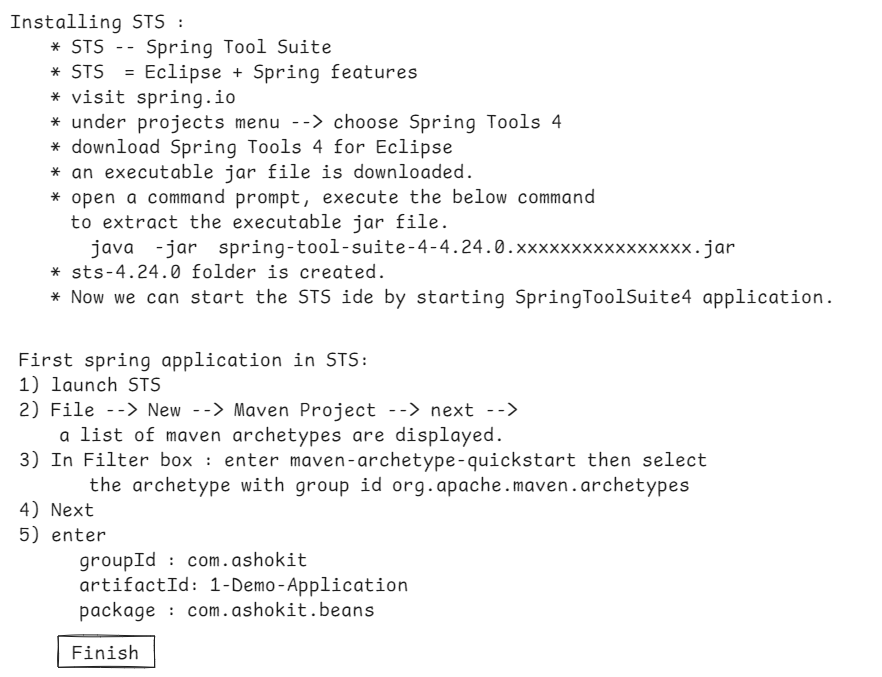
</dependency>











6) open pom.xml file, then add the below

dependency under <dependencies> tag.

<dependency>

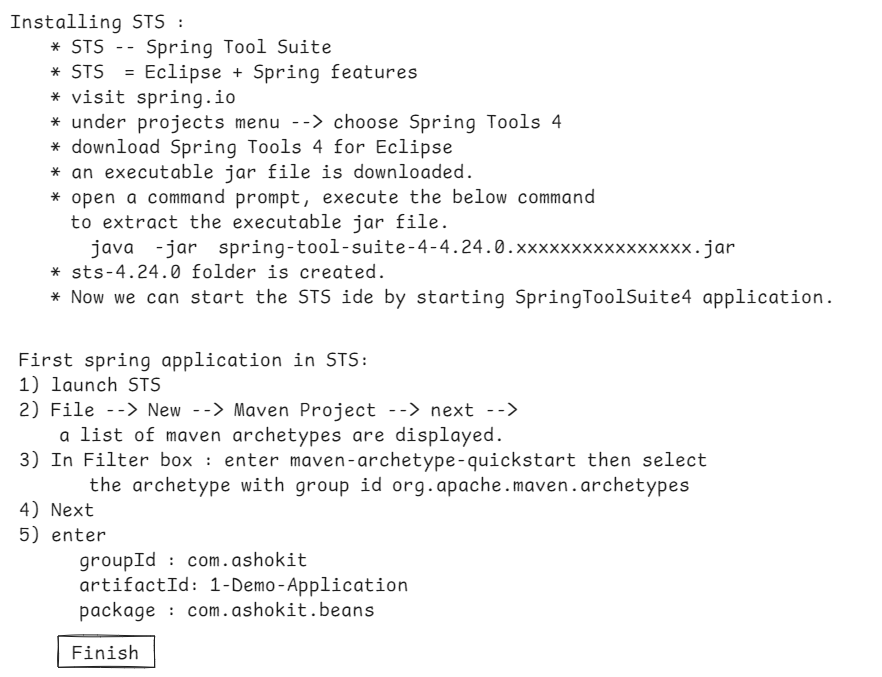
<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.3.37</version>

</dependency>





6) open pom.xml file, then add the below

dependency under <dependencies> tag.

<dependency>

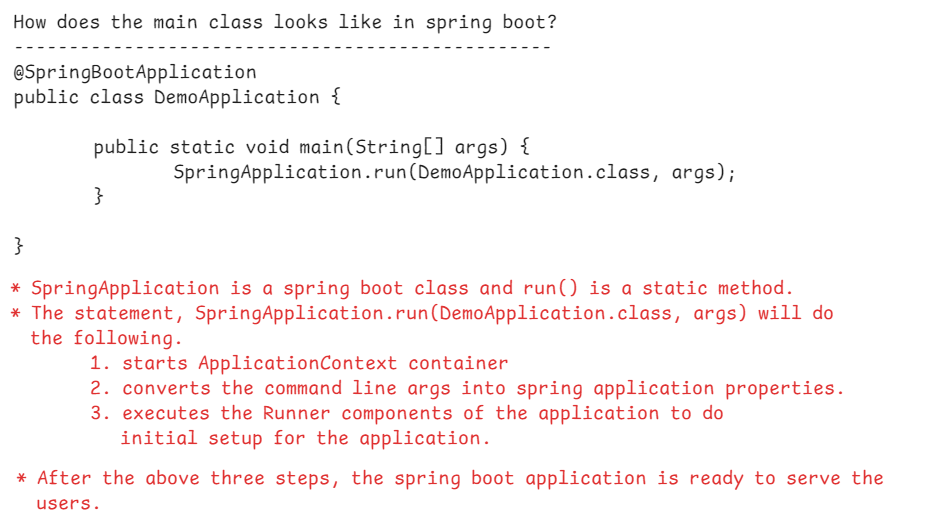
<groupId>org.springframework</groupId>

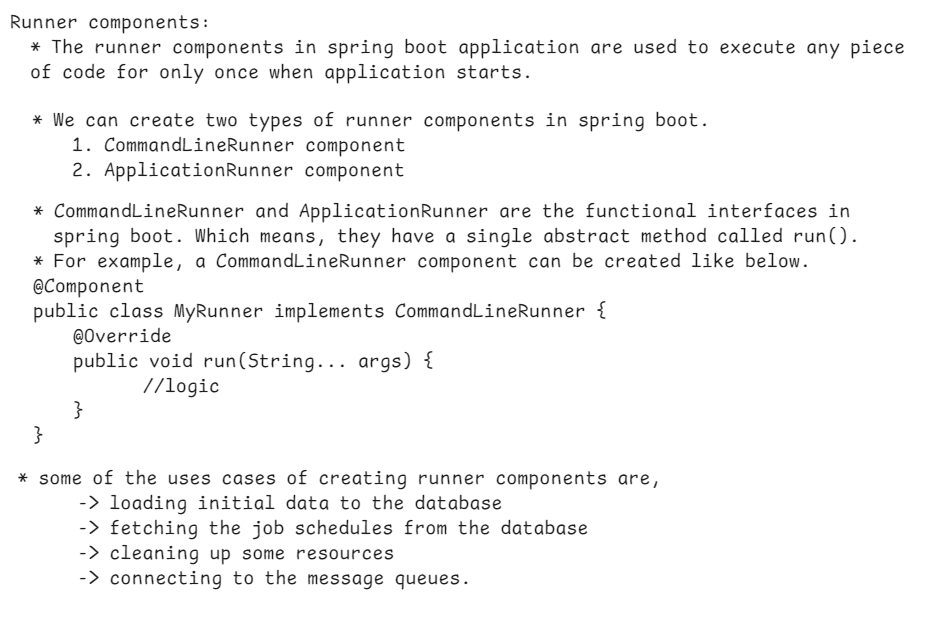
<artifactId>spring-context</artifactId>

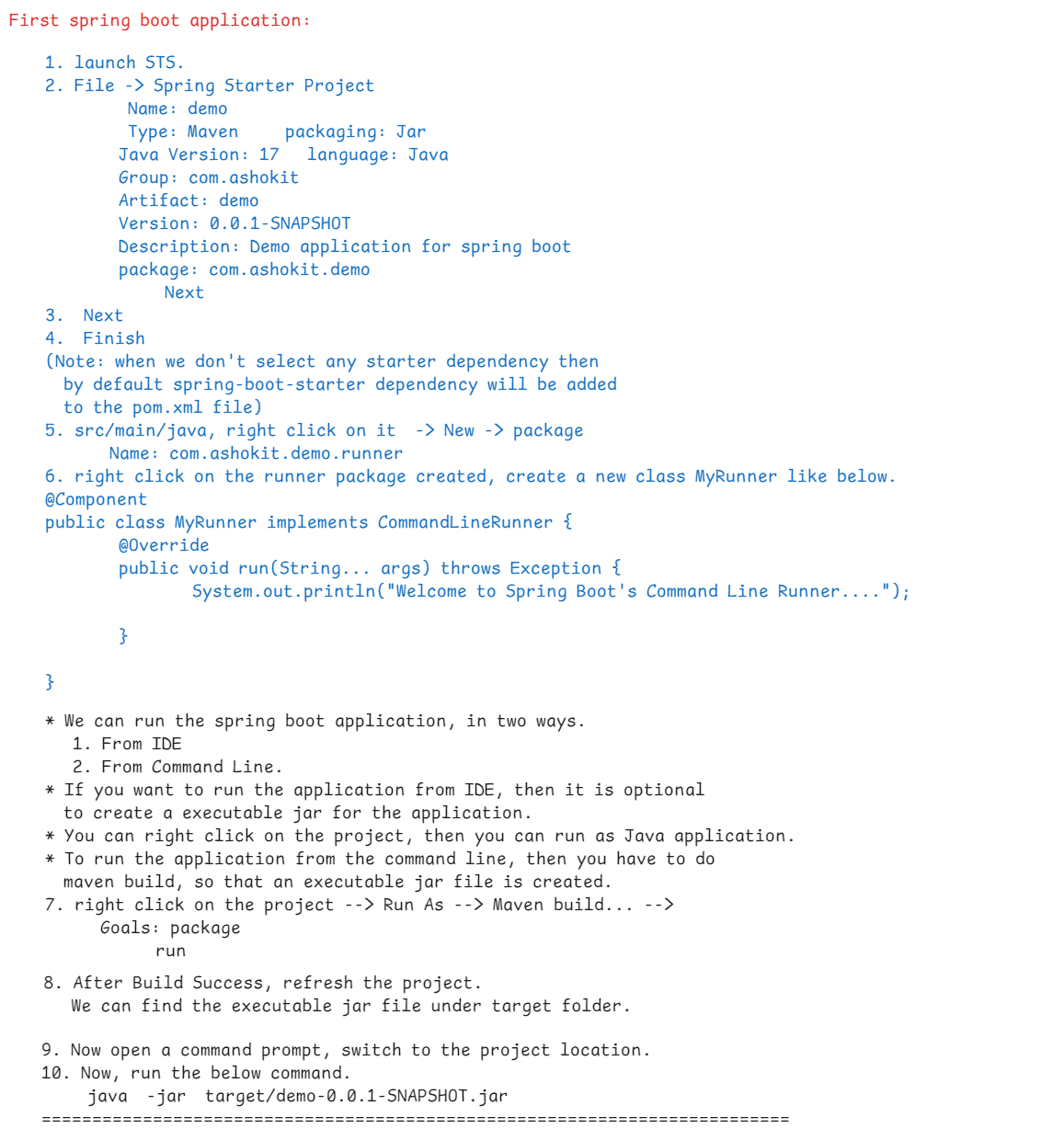
<version>5.3.37</version>

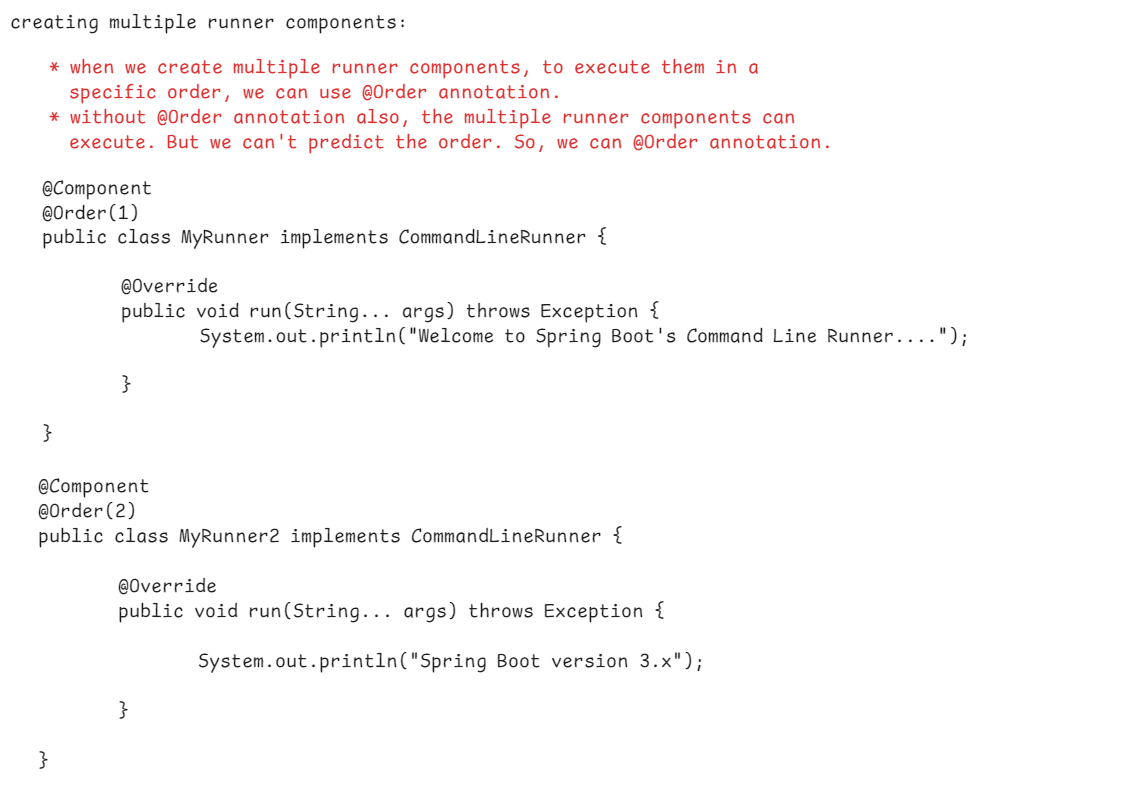
</dependency>











Working with properties file:

* When we create a spring boot application, at src/main/resources folder, application.properties file is created.
* spring boot will by default load the configuration properties from application.properties file into spring environment.
* a property can be injected to a bean property with @Value annotation.

for example:

application.properties:

----------------------

app.greeting=Welcome to Spring Boot

GreetingComponent.java

----------------------

@Component

public class GreetingComponent {

@Value(“${app.greeting}”)

private String greetingMessage;

public String greet() {

return greetingMessage;

}

}

* if the place holder in @Value annotation doesn’t match with a key in properties file then IllegalArgumentException will be thrown.
* So, we can specify the default value with a place holder in @Value annotation like below.

@Value(“${app.greeting:Welcome}”)

private String greetingMessage;

* if the file application.properties is renamed to some other name like myapp.properties then we must add @PropertySource annotation at main class.

ex:

@SpringBootApplication

@PropertySource(“{classpath:myapp.properties}”)

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

* if multiple properties files are created, say myapp.properties and myapp2.properties then the files can be loaded with @PropertySources annotation.

ex:

@SpringBootApplication

@PropertySources( {

@PropertySource(“{classpath:myapp.properties}”),

@PropertySource(“{classpath:myapp2.properties}”)

}

)

public class DemoApplication {

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

* If a matching key found in the two properties files then the value from second file will override the value from the first file.

command line arguments in spring boot:

1. option arguments
2. non-option arguments

* command line arguments are used to provide environment specific settings to the application from the execution line.
* option arguments are prefixed with a dash or double dash.
* option arguments are key=value pairs.
* non-option arguments are positional arguments and they do not start with a dash or double dash.
* non-option arguments are main inputs to the application and they do not have a key.

ex:

>java -jar target/demo.jar –config=config.yml output.log

(option argument) (non-option arg)

* Basically option arguments are used to modify the behaviour of an application or to provide some additional parameters to an application.
* non-option arguments are used to provide the primary data to the application.
* option arguments can be accessed easily with the key associated with a value.
* non-option arguments can be accessed based on their position.

Accessing command line arguments:

* CommandLineRunner interface provides run() method with String… args parameter. This parameter is to access command line arguments.
* The parameter which contains the raw command-line arguments as a simple string array.
* It's suitable for simple scenarios where you need to work with raw command-line arguments directly.

@Component

public class MyCommandLineRunner implements CommandLineRunner {

@Override

public void run(String... args) throws Exception {

System.out.println("CommandLineRunner - Arguments: ");

for (String arg : args) {

System.out.println(arg);

}

// Additional initialization logic

}

}

**ApplicationRunner interface**:

* Both CommandLineRunner and ApplicationRunner are interfaces in Spring Boot that are used to execute code after the application context is loaded and before the Spring Boot application starts.
* They are quite similar but have a difference in how they handle the command-line arguments.
* ApplicationRunner takes an ApplicationArguments parameter, which provides additional functionality over String... args. It allows you to retrieve both the raw arguments and the parsed options (i.e., non-option arguments and option arguments with values).

@Component

public class MyApplicationRunner implements ApplicationRunner {

@Override

public void run(ApplicationArguments args) throws Exception

{

System.out.println("ApplicationRunner - Arguments: ");

for (String arg : args.getNonOptionArgs()) {

System.out.println("Non-Option Arg: " + arg);

}

for (String option : args.getOptionNames()) {

System.out.println("Option Arg: " + option + " with values: " + args.getOptionValues(option));

}

// Additional initialization logic

}

}

* **CommandLineRunner** is best for simple scenarios with straightforward command-line arguments.
* **ApplicationRunner** is ideal for more complex scenarios where you need to handle and parse command-line options and arguments in a more structured way.

|| Date: 22-Aug-24 ||

Spring Boot JDBC

-------------------------

* 3 things to consider are,

1. data
2. medium
3. storage

raw data---🡪 I/O streams ---🡪 File

(data) (medium) (storage)

Java object -🡪 serialization -🡪 File

(data) (medium) (storage)

raw data --🡪 JDBC ---🡪 Database

(data) (medium) (storage)

Java object --🡪 JDBC/ORM ----🡪 Database

(data) (medium) (storage)

JDBC Vs Spring JDBC:

1. In JDBC, we have to write the boiler-plate code like loading driver, creating connection, creating statement, closing connection. But In Spring JDBC, spring will do this.
2. In JDBC, we must do exception handling. But in Spring JDBC, it is optional.
3. In JDBC, there is a chance of memory leak. But in Spring JDBC, no chance of memory leak.

|| DATE: 23-Aug-24 ||

JdbcTemplate class:

* This is the main class in Spring JDBC API.
* A spring bean, can perform CRUD operations on database, by using JdbcTemplate class.
* JdbcTemplate class depends on DataSource object,

to obtain a database connection.

* To obtain a databae connection, a java application has two options.

1. DriverManager class
2. DataSource object

* DataSource is an interface, it has multiple implentations, and the spring framework provided implementation is, DriverManagerDataSource class.
* Suppose, if we are creating a spring jdbc application, then the below changes are required.
* 1. add spring-jdbc dependency
* 2. add spring-context dependency
* 3. add jdbc driver dependency
* 4. configure/register DataSource object into the container
* 5. register JdbcTemplate object into the container.
* suppose, if we are creating a spring boot jdbc application, then the above configurations are not required.
* we need to add spring-boot-starter-jdbc dependency and jdbc driver dependency in pom.xml file.
* In application.properties file, we need to configure data source properties.
* Some methods of JdbcTemplate class, to perform a SQL operations are,

1. update(sql): performs insert/update/delete operation with the given static sql.
2. update(sql, args): performs insert/update/delete operation with the given dynamic sql, binding the args with the sql.
3. queryForMap(sql): performs select operation to select a single row, with the given static sql.
4. queryForMap(sql, args): performs select operation to select a single row, with the given dynamic sql, binding the args with the sql.
5. queryForList(sql): performs select operation to select mulitple rows, with the given static sql.
6. queryForList(sql, args): performs select operation to select multiple rows, with the given dynamic sql, binding the args with the sql.

* while selecting a single record with queryForMap() method, the selected record will be mapped to a Map object. So queryForMap() method returns a Map object.
* while selecting multiple records with queryForList() method, each record will be mapped to a Map object and the map objects are stored into a List. So queryForList() method returns a List object.

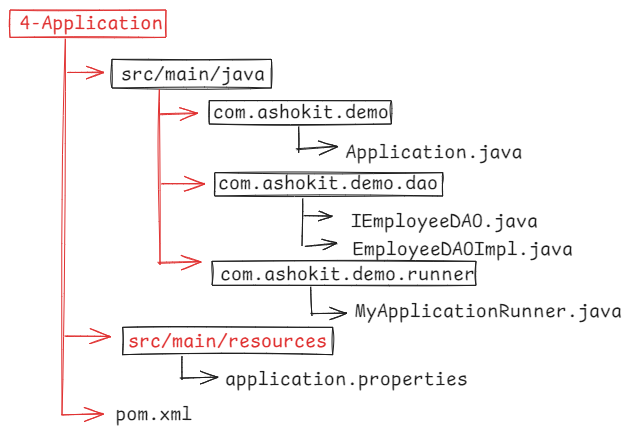
for ex:

Map<String,Object> map = jdbcTemplate.queryForMap(“SELECT \* FROM EMP WHERE EMPNO=7101”);

List<Map<String,Object>> lst = jdbcTemplate.queryForList(“SELECT \* FROM EMP”);

===============================================

|| DATE: 24-Aug-24 ||



Application.java

@SpringBootApplication

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Application.**class**, args);

}

}

IEmployeeDAO.java

------------------

**package** com.ashokit.demo.dao;

**public** **interface** IEmployeeDAO {

**void** createRow(**int** empno, String ename, **double** salary);

**void** updateRow(**int** empno, **double** salary);

**void** fetchByEmpno(**int** empno);

**void** fetchAll();

}

EmployeeDAOImpl.java

---------------------

@Repository

**public** **class** EmployeeDAOImpl **implements** IEmployeeDAO {

@Autowired

JdbcTemplate jdbcTemplate;

@Override

**public** **void** createRow(**int** empno, String ename, **double** salary) {

jdbcTemplate.update("INSERT INTO EMP VALUES(?, ?, ?)", empno, ename, salary);

System.***out***.println("Row inserted.....");

}

@Override

**public** **void** updateRow(**int** empno, **double** salary) {

jdbcTemplate.update("UPDATE EMP SET SAL = ? WHERE EMPNO = ? ", salary, empno);

System.***out***.println("Row updated......");

}

@Override

**public** **void** fetchByEmpno(**int** empno) {

Map<String, Object> empMap = jdbcTemplate.queryForMap("SELECT \* FROM EMP WHERE EMPNO = ?", empno);

Set<Entry<String, Object>> entries = empMap.entrySet();

entries.forEach( entry -> System.***out***.println(entry.getKey() + " - " + entry.getValue()));

}

@Override

**public** **void** fetchAll() {

List<Map<String, Object>> empList = jdbcTemplate.queryForList("SELECT \* FROM EMP");

**for** ( Map<String, Object> empMap : empList) {

Set<Entry<String, Object>> entries = empMap.entrySet();

entries.forEach( entry -> System.***out***.println(entry.getKey() + " - " + entry.getValue()));

}

}

}

MyApplicationRunner.java

@Component

**public** **class** MyApplicationRunner **implements** ApplicationRunner {

@Autowired

IEmployeeDAO empDao;

@Override

**public** **void** run(ApplicationArguments args) **throws** Exception {

empDao.createRow(7655, "JOHN", 5000.0);

System.***out***.println("====================================");

empDao.updateRow(7788, 9000.0);

System.***out***.println("====================================");

empDao.fetchByEmpno(7201);

System.***out***.println("====================================");

empDao.fetchAll();

}

}

application.properties

----------------------

spring.application.name=4-Application

# trun off the banner

spring.main.banner-mode=off

# data source properties

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=root