Spring Framework

It is a Java Framework which helps to create applications quickly by providing features that help you to reduce many repeated things in the project

You can use Spring Framework to develop many backend applications like

* Web applications: Front end & back end both are developed in spring
* Cloud based applications: Backend application that you can deploy on cloud machines like AWS, Azure, Google cloud etc (spring boot & microservices help you to do this)
* Webservice based applications: Backend application that helps many front end applications to interact like React, Angular, Vue etc.

Spring modules: These are libraries which help you to create applications quickly

1. Spring Core: Base module for any spring modules, because it gives you all the design patterns required to quickly create application
2. Spring MVC: Module which is to develop web applications & REST based webservices
3. Spring JDBC: Module to interact with database
4. Spring Boot: Module to quickly configure spring applications in less time, earlier spring configurations were taking more time, but because spring boot its very quick
5. Spring Cloud: Module to develop applications for cloud environment

Spring Core:

It uses Dependency Injection feature, where an object is supplied into another objects, i.e., an object which depends on another object is created by spring container, this process is also called as Inversion of Control (IOC)

IoC: here the process of creating object is reverse hence its called as Inversion of Control

Dependency Injection between the objects with simple & complex types

You can have setter methods to supply the dependencies of simple or complex type

class MySqlDBImpl implements DBOperation{   
 private String dbUrl ; // simple type   
}

class UserService {   
 private DBOperation dbop;  
}

<bean id = “b1” class = “com.crisil.MySqlDBImpl”>  
 <property name = “dbUrl” value = “jdbc:mysql://localhost:3306” />  
</bean>  
<bean id = “b2” class = “com.crisil.UserService”>  
 <property name = “dbop” ref = “b1” />  
</bean>

Spring Jdbc (Java Database Connectivity)

It helps you to interact with any relational database systems like Oracle, MySQL, Derby, Sybase and so on

It gives you a configuration class (DriverManagerDataSource) to configure datasource information like username, password, url, driverClass, this class establishes database connection object for you using the configurations done in XML

It gives you a template class (JdbcTemplate) that uses DriverManagerDataSource to perform CRUD operations

DriverManagerDataSource: It has 4 properties

1. username
2. password
3. url
4. driverClassName

JdbcTemplate: It has a property

1. datasource: must reference DriverManagerDataSource

Steps

1. Derby Database configuration with tables
2. Project with dependencies - spring-context, spring-jdbc, derby-client
   1. Employee class - id, name & salary
3. interface EmployeeDao
4. Implementation EmployeeDaoImpl
5. Employee Service class
6. XML Configuration
   1. DriverManagerDataSource - username, password, url, driverClassName
   2. JdbcTemplate - datasource
   3. EmployeeDaoImpl
   4. EmployeeService
7. Main class that acts like Controller

Implementing EmployeeDaoImpl

JdbcTemplate gives you methods to perform CRUD operations

1. update(sqlQuery, args)
2. queryForObject(sqlQuery, rowMapperObject, Object…)
3. query(sqlQuery, rowMapperObject)

update: It is used to run insert, update & delete types of query

queryForObject: It is used to run select query that can return one result, mainly used when you want an object with a particular id

sqlQuery = “select \* from employee where id = ?”

Object… = it accepts value for ‘?’

query: It is also used to run select query but it returns multiple records, mainly used when you want multiple objects for a query

rowMapperObject: It is a class that implements RowMapper interface that takes care of converting sql records to java object, it has one method map(int rowNum, ResultSet rs)

When select query is triggered a one or more rows will be returned which is handled by ResultSet & rowNum gives index number for each ResultSet, on each map() invocation we can expect one java object

Spring ORM

ORM stands for Object Relational Mapping, which helps you to directly map java objects to the database table, it gives you the feature where you can directly store java objects and retrieve java objects without writing any SQL query

Features of ORM

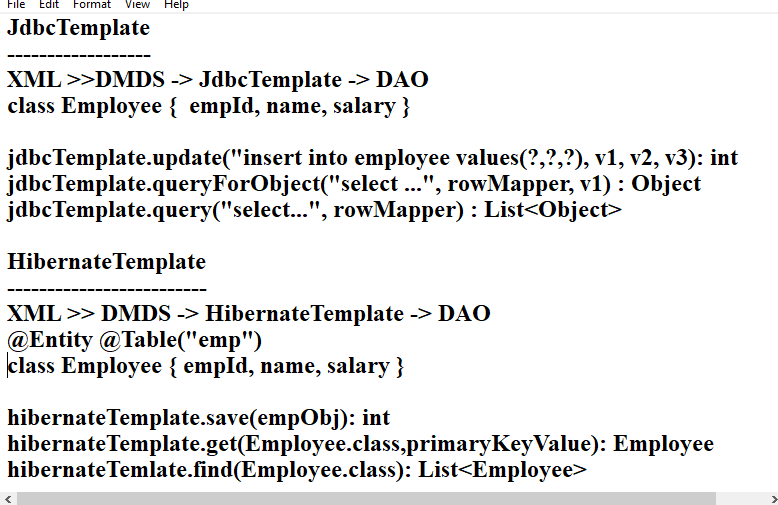
* Provides inbuilt methods to store, update, delete & retrieve Java objects without SQL query
* Database table & Columns names you don’t have to use while perform crud operations
* Entity class are used to represent the table information for the java object, this helps to recognize the table names & columns names while performing CRUD operations
* You will perform CRUD operation on the entity class object

HibernateTemplate: It is an inbuilt class that has methods to perform CRUD operations

Spring JDBC vs Spring ORM

Both are used to interact with the database but Spring ORM has some advantages

|  |  |
| --- | --- |
| Spring JDBC | Spring ORM |
| JdbcTemplate is used to perform operations | HibernateTemplate is used to perform operations |
| JdbcTemplate uses SQL query which are database dependent | HibernateTemplate uses inbuilt methods that takes care of generating SQL query based on the underlying database |
| If DB changes you need to change the query written in the code | If DB changes you don’t need to change the code because you don’t write SQL query at all |
| Table & Column names are used in the query | Table & Column names are used in the entity class, the entity class object & its properties recognize the table & column details |
| You need to take care of converting SQL records to Java objects & vice versa i.e., RowMapper we used to convert SQL records to Java objects | HibernateTemplate methods will take care of converting java to sql records and vice versa, you don’t need any RowMapper here |



Spring Boot

It simplifies spring configurations in the application automatically with the inbuilt spring boot starter libraries, you don’t need to have XML file at all, any application related configuration you need can be configured in a property file ‘application.properties’ in a simple key value configuration, spring boot takes care of configuring the application based on the inbuilt spring boot starter libraries

Some of the starter libraries

1. Web: This does all the configuration required for web & rest applications like server configuration, front controller configuration, scanning all the @Component, @Service, @Repository, @RestController classes to create their objects in the spring container
2. Data JPA: This does all the configuration required for database beans like DriverManagerDataSource injecting to JdbcTemplate, HibernateTemplate & so on, it automatically reads application.properties for datasource informations like username, password, url, driverClassname, etc.
3. Spring Cloud: This does configruations required for cloud environments like service discovery, load balancer and etc

How spring boot automates the configuration

It uses @SpringBootApplication annotation which takes care of autoconfiguring the application based on the library you have in the project, a class must have @SpringBootApplication and this class must be loaded to trigger the automation

@SpringBootApplication  
public class AppStarter {   
 public static void main(String[] args) {   
 SpringApplication.run(AppStarter.class); // triggers automation  
 }  
}

The moment AppStarter is loaded it looks for all the libraries in the project & also application.properties to perform the configuration required

Note: By default all the classes of your project are scanned by looking the sub-package of the class having @SpringBootApplication, hence you must have all the classes in the sub-package of the class having @SpringBootApplication

@Autowired: It is used to supply the dependency to an object

@Component, @Service, @RestController, @Repository: These annotations are written on top of the class which lets spring to create the object and maintain in the spring container

@Component   
class A { }   
@Component   
class B {   
 @Autowired  
 private A obj; // no need of setter method   
}

@Service  
class C {   
 @Autowired  
 private B objB; // object of B is injected to C object  
}  
All the objects like A, B & C are created in spring container

RESTful Webservices

ReST stands for Representational State Transfer which is used to make heterogenous applications to exchange the data regardless of the technology they are using, ReST webservices uses Http methods to specify the operations each Http methods must have some URL’s which is used by the applications who wants to communicate

Http methods

1. HTTP Post: To perform store/create operation
2. HTTP Get: To perform retrieve/fetch operation
3. HTTP Put: To perform update/modify operation
4. HTTP Delete: To perform remove/delete operation

These 4 are the common methods used by applications in their end to communicate

URL

Every operation you perform must have an URL which is used by another application to communicate

Creating ReST webservices using Spring Boot

We need to use spring boot starter web library to create ReST webservices because it gives all the configurations required for web applications & ReST webservices like

* Server configuration
* Component scanning - @Component, @Service, @RestController and etc
* Dependency Injection using annotations like @Autowired
* application.properties - for some configurations

Spring Initializr: It is a website by spring that helps you to quickly get a spring boot project template

Changing the port number of server

You must use application.properties file with a key called server.port

server.port = 9090

Spring Data Jpa

It will automate performing the CRUD operations on any entity class, it provides some interfaces like

* CrudRepository<T>
* JpaRepository<T>

These interfaces you must extend with the interface you want in the DAO layer and spring boot will automatically implement the interface based on the entity class you use

interface EmployeeRepository extends JpaRepository<Employee, Integer> { }

Here Employee is an Entity class mapped to employee table, EmployeeRepository gets all the methods of JpaRepository like save(object), deleteById(id), findAll(), getById(id) and so on

Note: Spring Boot will take care providing a proxy implementation for the EmployeeRepository so that all the methods would work on the table without any problem

We need to only inject the EmployeeRepository to the @Service class

Important methods of JpaRepository and their signature

1. save(T obj): T
2. deleteById(T primaryKey): void
3. findAll(): List<T>
4. findById(T primaryKey): Optional<T>

Setup for spring boot & spring data jpa

1. Add Spring Data Jpa & Derby library
   1. spring starter data jpa
   2. derby client
2. Create an entity class that maps to employee table
   1. Start the database server (startNetworkserver)
3. Create an interface in dao layer that extends JpaRepository<T, ID>
4. Create a class in service layer that injects dao layer object
5. Configure application.properties with datasource like username, password, url, driverClassName
6. Update the Rest Controller class to perform the operations defined in the service layer class

Activity:

1. delete by id
2. update salary by id

Implementing delete by id in service & controller

Service:

@Transactional  
public void deleteEmployee(int id) {   
 dao.deleteById(id);   
}

Controller  
@DeleteMapping(“/{id}”)  
public ResponseEntity<Object> delete(@PathVariable(“id”) int id) {  
 service.deleteEmployee(id); // rest of the code  
}

Implementing update salary by id in service & controller

Service:

@Transactional  
public Employee updateEmployee(int id, double salary) {   
 Employee e = getEmployee(id);   
 e.setSalary(salary);   
 dao.save( e ); // this is optional  
}

Controller

@PutMapping(“/{id}/{salary}”)  
… update(@PathVariable(“id”) int id, @PathVariable(“salary”) double salary) {  
 // call the updateEmployee by passing id & salary  
}

Creating custom exception

public class EmployeeNotFoundException extends Exception {   
 public EmployeeNotFoundException(String err) { super (err) }  
}

throw new EmployeeFoundException(“id not found”) => caller handles this exception and can use getMessage() of exception class to know the message

catch(EmployeeNotFoundException e) {   
 e.getMessage(); //   
}

CORS:

Cross-Origin Resource Sharing is disabled by default at the backend if any front-end needs to access hence you must enable which front-end can access the backend by configuring the CORS at the backend application.

Spring uses an annotation called @CrossOrigin(origin = { }) which allows you to configure all the origins who can access the application.

If angular which is running in <http://localhost:4200> then we must add this origin in @CrossOrigin as below

@CrossOrigin(origin = { “<http://localhost:4200>” } on top of the controller

Integrating Angular with Spring Boot Services

Steps:

1. Add @CrossOrigin in spring boot webservice
2. Create angular project & select yes for routing, select CSS for styling (name of the project: employee-ui)
3. Ensure Database, Spring Boot application is running
4. Add bootstrap in the angular
5. Add FormsModule, ReactiveFormsModule, HttpClientModule in AppModule of app.module.ts
6. Create components for storing, fetchAll, update
7. All the components you create can be shown in root component or using routers
8. Create an angular service and inject HttpClient & return all the methods with Observable
9. use subscribe({ }) to get the data from the response

Adding routerLink so that you can load the components based on the links instead of showing all the components in a single root component

<a routerLink = “/store”>Store</a> |   
<a routerLink = “/update”>Update</a> |  
<a routerLink = “/findAll”>Find All</a> |  
<a routerLink = “/find”>Find</a> |

These links must load components which are configured in the router i.e, app.routing.module.ts

const routes : Routes[] = [  
 {path : ‘store’, component: StoreComponent },   
 {path: ‘update’, component: UpdateComponent},  
 ….  
]