# Spring

1. Spring Framework is highly based on configuration, which is why it was developed because back in early 2000s making a Java Enterprise application was tough because of the complexity, the framework then present had a steep learning curve and was inflexible.
2. With Spring POJOs were used to replace EJB(Enterprise Java Beans) with simpler java objects that were easy to make, test, deploy.
3. It is non-intrusive, only required parts of framework can be used rather than the whole. Implementing IOC which was implemented by the framework where the bean creation and injection would be handled by the framework increasing modularity.
4. IOC: Inversion Of Control is a philosophy which says we should not worry about object creation, the object will be created for us. Where Spring handles object creation and manages the lifecycle of object. This is handled by external framework like Spring.
5. Dependency Injection: IOC is implemented in Java using DI. @Value is also example of di although it is a simple injection.
   1. Constructor. Injection is default, no need to write @Autowired. If multiple constructors then needed. <constructor-arg value/ref=””> inside <bean id=”” class=”” > for xml.
   2. Setter. @Autowired annotation required. <property name=”” value=””> for xml.
   3. Field Injection. @Autowired annotation required. To use @Autowired in xml config add <context: annotation-config/>
      1. Not recommended because unclear which dependencies are required.
      2. Creating mocks or implementing tests becomes tricky because the framework handles injections.
      3. Dependencies are injected after the object is created, leading to the possibility of using an object before it has acquired all required dependencies.
      4. IOC promotes explicit dependency management, but dependencies are injected behind the scene.
   4. DI on the object happens based on data types.
      1. If there are two same data types variables then if spring gets confused while injecting them, it throws NoUniqueBeanDefinitionException.
      2. Primitive Type Dependencies: like int,boolean
      3. Collection Type: like list,set,map
      4. Reference Types: to other beans
6. Annotations reduces XML configuration overhead.
7. SpringContainer
   1. Contains Beans, config files, aop files, dependencies.
   2. Uses XML to know which object needs which properties where It needs to be injected.
   3. BeanFactory and ApplicationContext.
   4. ApplicationContext interface represents IOC container.
      1. ClasspathXMLApplicationContext give classpath for the xml config.
      2. FileSystemXMLApplicationContext to give file location for the xml.
      3. AnnotationConfigApplicationContext is used when beans are configured using annotations like @Configuration. In that appconfig class, we need to make methods to return objects of the bean types. And annotate @Bean over the methods.
8. @SpringBootApplication
   1. @EnableAutoConfiguration. Automatically configures application based on the dependencies in classpath. Meaning, we add dependencies in our project, they are automatically downloaded and added to the classpath. Spring goes through this classpath and looks which dependencies are present automatically configuring them. Ex: SpringWeb, a tomcat, a dispatcherServlet is automatically configured, even default error pages, ensuring speedy project building.
   2. @ComponentScan(basePackages={}). Scans the package where the main class resides for spring components, @Component, @Service, @Repository, @Controller.
   3. @SpringBootConfiguration, it extends Configuration so working is same. But specific for spring boot projects, AppConfigs can be marked with it, where beans are returned. Indicates the class contains Spring Configuration Methods, marked with @Bean.
9. SpringApplication.run(class,args) return an ApplicationContext type.
   1. We don’t need to create an object of SpringApplication, run is a static method.
   2. But sometimes like stopping the display spring banner we need to create an object of springApplication because that method is non static. Proves SpringApplication itself is not static but has some static methods like run(args). App.setBannerMode(Banner.Mode.OFF);
   3. This statement starts the IoC Container or Spring Container inside the JVM which holds all the objects that spring creates.
   4. Using this ApplicationContext the objects can be accessed.
   5. ApplicationContext context=SpringApplication.run(class,args); Class ob=context.getBean(class); then the method of that class can be accessed.
10. Spring does not create objects for all the classes. @Component asks spring to create an object for it.
11. @Autowired knows which bean to map inside the setter method or field injection by checking the type.
    1. If an interface has a parent type, and two beans subclasses inherit them. Then spring will be confused which to inject because two beans with same type.
    2. Then @Primary is used to select one over other.
    3. @Qualifier(“class name in lowercase”) under @Autowired lets spring know which bean we are looking for.
    4. Beans name comes from the class. Laptop class bean name is laptop.
12. Different types of configurations:
    1. Java class based.
       1. ApplicationContext c=new AnnotationConfigApplicationContext(AppConfig.class);
       2. @Configuration on this class.
       3. To create a bean of class Desktop. @Bean public Desktop desktop(){return new Desktop();} return an object. Spring will manage and inject it.
       4. @Bean(name={“different”,”bean”,”names”}). Normally the method name is the bean name.
       5. To autowire a bean, public Alien alien(@Autowired Computer com){obj.setCom(com);} Autowired is optional. Computer is an interface. Desktop is an implementation, and its bean is created. Spring figures out and injects Desktop into Computer.
       6. If Computer has multiple implementations like Desktop and Laptop. Then public Alien alien(@Qualifier(“desktop”) Computer com) needs to be added, which solves the ambiguity. Or @Primary is added on the bean.
    2. XML
       1. ApplicationContext c=new ClassPathXmlApplicationContent(xml file); Dev o=(Dev)c.getBean(“dev”); o.method();
       2. This is xml config file usage.
       3. it is verbose, follows “configuration outside code”.
       4. Beans are explicitly defined.
       5. Ideal for applications that need frequent configuration changes without redeploying the application.
       6. Suitable for legacy projects.
       7. Errors discovered only on Runtime.
       8. <bean></bean> defines beans.
       9. <property name=”” value=””/> inside bean sets the property of a value using setter method.
       10. Constructor Injection:
           1. Implicit: when all the params are different types.
           2. Explicit:
              1. Using index: <constructor-arg index=”” value=””/>
              2. Using type: <constructor-arg type=”” value=””/>
    3. Annotations
       1. Concise, compact.
       2. Configuration is embedded in code making it easier to manage without switching between XML files and java classes.
       3. Errors are checked in compile time.
       4. @Component added on classes. @Component(“beanName”)
       5. Inside AppConfig class, @Configuration, @ComponentScan(“packages”)
       6. ApplicationContext c=new AnnotationConfigApplicationContext(AppConfig.class);
       7. @Qualifier(“className”) when wanted 1 found 2 class error.
       8. @Primary. Used together, @Qualifier is given preference over @Primary.
       9. @Scope(“prototype”)
       10. Inject values. @Value(“5”) private int x;
13. ApplicationConfig class:
    1. Used as a replacement to XML based configuration. Configure the application’s beans and other settings.
    2. @Configuration written above the class.
    3. @ComponentScan(basePackages=””) if multiple base classes basePackages={“”,””} @ComponentScan(“package”) also works.
    4. Used to create beans of the the different classes and return them.
14. Driver class:
    1. ApplicationContext context=new AnnotationConfigApplicationContext(ApplicationConfig.class); return context.getBeans(ClassName.class);
    2. In main method this object is used to call methods of that class.
15. All classes are marked as @component, the ones object need to be created. In the component there should be getter and setter/ constructor for setting the value of the variables.
16. @PropertySource(“classpath:goldRateDetails.properties”) added in Config Class.
    1. Values defined inside the properties file can be used like
    2. @Value(“${goldValue}”)
17. If ApplicationConfig Class is not created.
    1. In Driver.java write
    2. @Configuration
    3. @ComponentScan
18. @Value
    1. (“Injects hard coded string”) the type typecasted according to the variable.
    2. (${…}) using placeholder syntax. These files are typically included in the classpath.
    3. (#{…}) called Spring Expression Language. SpEL allows insertion of dynamic expressions to calculate or manipulate injected values.
    4. @Value("#{${rates}}") rates={1:100.0, 2:200.0} used to insert into maps.
    5. @Value("${server.list}") server.list=server1,server2,server3 insert into List<>.
19. @Scope(“prototype”)
    1. Creates new instances everytime it is accessed.
    2. Generally, beans are singleton in nature. One instance is created and shared across the application.
20. @Bean
    1. Is used to create a bean in IoC Container.
    2. In the application config, beans can be initialized by calling their constructor and returning the object.
21. @PreUpdate: lifecycle callback in JPA
    1. The method annotated with this, will run before any existing data is updated inside db.
    2. Used for auditing purposes.
    3. Validating entries are correct.
22. @PrePersist: lifecycle callback in JPA
    1. Before the entity is inserted into db, to fill data for some columns who don’t have a default value, used. Before inserting any entity it runs.

XML file

1. <?xml version=”1.0” encoding=”UTF-8”?>
2. <beans xmlns schema/util/p/context and schemaLocations are provided(These are called namespaces which are to be used then)></beans>
3. <bean name=”” class=””></bean>
4. Injecting values using getter-setter methods.
5. Inject values in the bean
   1. <property name=”” value=””>
   2. <property><value></value></property>
   3. Using p-schema. <bean p:id=”20” p:name=””>
6. Injecting Collections in the bean
   1. <list> <value></value> </list>
      1. If list has only one element then no need to write <list></list>
   2. <set>
   3. <map><entry key=”” value=””>
      1. <entry key="”><value></value>
      2. <entry value=””><key></key>
   4. <property name=””><props><prop></prop>
7. Injecting References:
   1. <bean class=”B” name=”b”/>
   2. <bean class="A” name=”a”> <property name=”b”><ref bean=”b”/></property></bean>
   3. <bean class="A” name=”a”> <property name=”b” ref=”b”/>
   4. <bean class=”A” name=”a” p:scores-ref=”b”/>

Life Cycle Methods

1. public void int() is executed after the value is inserted into the object or object is instantiated.
2. destroy() used before the destruction of object.
3. Using XML Configuration:
   1. Create methods in the class and configure them.
   2. In xml <bean init-method=”” destroy-method=””/>
   3. Spring calls the setter method, then init method.
   4. To run the destroy method, hook needs to be added. ApplicationContext does not have the hook but its parent class AbstractApplicationContext has it. ctx.registerShutdownHook().
4. Using Spring Interfaces:
   1. Class implements InitializingBean{}
   2. afterPropertiesSet() is init method.
   3. Class implements disposableBean{}
   4. destroy() method.
5. Using Annotations:
   1. javax.annotation-api dependency added inside pom.xml
   2. @PreConstruct for init method
   3. @PreDestroy for destroy method
   4. CommonAnnotationBeanPostProcessor is used to enable these annotations. Default, annotations are disabled. <bean class=”…commonannotation…”/>
   5. <context:annotation-config /> enables all annotations.

Dependency Check:

1. @Required used on setter function makes dependency necessary. If bean is not provided during initialization then BeanInitializationException is thrown.

Scope

1. Singleton: one object is created and accessed everytime. Spring container creates a bean. if same type bean is created again it refers the old bean address. In singleton behavior. <bean scope=”singleton”>
2. Prototype: different objects are created everytime accessed. <bean scope=”prototype”>
3. Request: when a user requests a bean is created, multiple users multiple beans. Spring MVC. A new bean is created at the beginning of HTTP request. Bean is destroyed after the request ends.
4. Session: a bean for a user in one session. Spring MVC.
5. GlobalSession: used is springportlets one global object for all portlets.

Bean can be defined inside a bean creating inner bean.

Constructor value injection:

1. <constructor-arg><value></value> or <ref bean=””</constructor-arg>
2. Using c namespace. All same like p namespace.
3. On constructor overloading, there is ambiguity in which constructor should be called if exact constructor not found.
   1. Spring prefers exact match of type, if that does not happen it does implicit conversions. If 10.5 is there and no float/decimal constructor, and no explicit type conversion is also done. Then, compile time error.
   2. String constructor is taken first if found.
   3. Double or Int constructor whoever at beginning goes first.
   4. Because all values are normally taken as String then converted to the necessary type using parseType().
4. The ambiguity can be solved:
   1. <constructor-arg value=”10” type=”int”/>
   2. <type=”double/>
   3. index. To keep the order correct, use index when there are multiple values.
   4. name. to use the name of the variable.

Reading properties

1. database.properties
2. <context:property-placeholder location=”database.properties”/>
3. <constructor-arg><value>${dbServer}</value></constructor-arg>
4. If a property does not exist in properties file but is used inside value exception BeanDefinitionStoreException is thrown.

Exception

1. BeanCreationException: bean could not be created.
2. NoSuchBeanDefinitionException: bean definition was not found.
3. UnsatisfiedDependencyException: spring cannot satisfy a dependency.
4. BeanNotOfRequiredTypeException: thrown when bean retrieved from appcontext not of required type.
5. BeanDefinitionStoreException: bean could not be parsed or loaded.

Autowiring

1. <bean .. autowire=”byType”/> will do di based on bean Type.
2. “byName” looks for beans with same name as the object reference name.
3. “constructor” looks for beans with same type defined in the constructor. If not written this, then automatic constructor based di will not happen. Then <constructor-arg ref=”bean”/> needs to be written inside the dependent bean.
4. If no bean is found it will insert null value.
5. Duplicate beans found then throws an exception.
6. @Autowired annotation over field/constructor/setter method. Enabling annotations by <context:annotation-config/>
7. If multiple beans of the same type are present, @Qualifier(“beanName”) is used.
8. @Autowired(required=false) injects a null value if no bean is found.

AOP

1. CrossCutting concerns like logging, security, exception handling or perfommance monitoring which are not dependent on specific classes. But is used by many classes.
2. Business classes can use external services without changing its code.
3. With a centralized approach its easier to debug.
4. Minimizes repetitive code by centralizing functionality, such as logging or security in a single aspect.
5. Aspect: External Services that are used.
6. Advice: Methods defined inside aspect class.
7. PointCut: Determines which methods in our application needs advice.
8. JoinPoint: Says which business method needs which advice.
9. Target: Object of the business class which needs advices.
10. Weaving: Process of applying the advices onto target using joinpoint. Done by special component weaver.
11. Proxy: Class generated due to Weaving process. Combination of business logic and advice logic.
12. Frameworks:
    1. Spring AOP
    2. AspectJ
    3. JBoss AOP

PointCut

1. Access Specifier Return type package/class methodName methodParams(..) throws
2. Pointcut syntax
3. execution(public void com.service.PService.addProduct(..))
4. execution(\* com.service.\*.\*(..))

AspectJ Annotation Driven

1. @Aspect class contains all advices and pointcuts.
2. @Before: method applied before a particular aspect method is invoked.
3. @After: method applied after a particular aspect method is invoked.
4. @AfterReturning: method applied after aspect method returns.
5. All three above use:
   1. Public void log(JoinPoint joinPoint)
   2. It can’t control the method execution but it can perform accordingly.
6. @Around: method applied after and before aspect.
   1. Object log(ProceedingJoinPoint joinPoint) throws Throwable{}: ProceedingJoinPoint has proceed() which allows to execute or skip the method.
   2. Object res=joinPoint.proceed(); runs the method.
   3. With around even the return from a method can be changed.
7. @AfterThrowing: if exception is thrown the advice is applied.
8. These are added in the aspect classes not in business logic.
9. <aop:aspectj-autoproxy/> will enable all aspect based annotations.

Spring Boot

1. Convention over configuration
2. Opinionated Defaults: boot is configured with sensible defaults based on common best practices.
3. No XML. It configures from the jars present in the class path.
4. POMs bring the jars. Inside the JARs, META-INF/spring.factories file is present which has all the autoconfiguration @Configuration classes and @Condition checks.
5. This simplifies application setup by automatically configuring what I need rather than manual setup.
6. Starter dependencies bring with itself many jars.
7. @SpringBootTest looks for class marked with @SpringBootApplication and use that to run the application.
8. @Service, @Repository
9. The file containing the main method with @SpringBootApplication is considered the entry point of the application.

JDBC Template

1. JDBC technology is a connector that connects JAVA to dbs + Template design pattern defines the skeleton of an algorithm, allowing subclass to implement specific things without changing the algo structure.
2. Handles closing connections, preparing statements and handling exceptions for you.
3. Inbuilt methods like update() and query() for result fetching.
4. Map db rows to objects using rowMapper.
5. JDBC template does not open the connection DataSource does.
6. Datasource manages the connection, everytime a connection is needed it is given in an instant, after query completes the connection is not destroyed, it is moved to the pool. So, without using datasource we need to create a connection manually everytime which is slow and heavy. Memory leaks could happen eventually.
7. Memory leaks: connection is opened, but never closed. So that portion of ram is never freed. Too many such instances will make the computer run out of memory.
8. Interface javax.sql.DataSource is written in JAVA standard API. It is implemented by Spring as DriverManagerDataSource. It takes args (driverClassName, url, username, password) and returns a datasource. The driverclassName specifies which jdbc driver to use to make the connection.
9. JDBC Template(dataSource is injected).
10. A bean is created of the jdbcTemplate. A sql statement is written and then jdbcTemplate.update(sql,values);
11. Data Source bean:
    1. driverClassName=com.mysql.jdbc.Driver
    2. url=jdbc:mysql://localhost/mydb(database name)
    3. username
    4. password
    5. name=dataSource
12. Jdbc Template:
    1. dataSource-ref=”dataSource”
    2. name=jdbcTemplate
13. Jdbc template internally responsible executing the preparedStatement.
14. DTO- carries the data.
15. DAO-the dto is accessed and inserted into the database, an interface is created with the methods.
16. The jdbcTemplate should be created inside the dao impl.
17. jdbcTemplate.update(String sql,id,firstname, all the values that the sql has using ?).
18. Select statement in jdbcTemplate has methods
    1. queryForObject(sql,rowmapper,object args) returns an object(One column, one row or mapped object)
    2. query(sql,rowmapper):List<>(multiple rows)
19. SQL query on running gives a table like structure called ResultSet. Which is used to map rows of data into java objects.
20. RowMapper is an interface in spring used for this. Override the mapRow method from that interface. Class EmployeeRowMapper implements RowMapper<Employee>{}
21. Public Employee mapRow(ResultSet rs,int rowNum) throws SQLException{Employee object is created. E.setId(rs.getInt(“id”)); e.setName(rs.getString(“name”)); return e;}
22. Executing select, rows are returned from the database in ResultSet, RowMapper is used then.

JDBC

1. Import packages(java.sql.\*), load driver, register driver, create connection, create statement, execute statement, close.
2. Connection con=DriverManager.getConnection(url,user,password);
   1. Connection is an interface. DriverManager gives an object of it.
3. url=jdbc:mysql://localhost:3306/mydb
4. Statement st=con.createStatement();
5. ResultSet rs=st.executeQuery(sql); returns a resultset.
6. st.execute(sql) will return Boolean.
7. while(rs.next()); returns Boolean if data is present. rs.getString(1);
8. con.close(); at the end.
9. PreparedStatement vs Statement:
   1. PreparedStatement precompiles queries making execution faster. Statement parses and compiles queries every time adding overhead.
   2. PreparedStatement used parameterized queries, preventing malicious input from altering SQL execution, Statement directly concatenates input into the query, making it vulnerable to SQL injection attacks.
   3. PS makes handling dynamic values cleaner without manual string concatenation, Statement requires string manipulation, increasing complexity.
10. PreparedStatement st=con.prepareStatement(sql);
11. st.setInt(1,sid); st.setString(2,sname); st.execute();

ORM

1. Every field in the class will be automatically mapped to a database column and vice versa, while fetching data or sending data. This saves time increasing productivity.
2. Converts object into table row and vice versa. No need for a query.
3. JPA used for doing ORM in Java EE. JPA has specifications and API. Hibernate follows JPA.
4. Session is an interface, SessionFactory is also an interface. Configuration is a class.
   1. Create an object of Configuration. cfg.configure().
   2. Use it to create a sessionFactory. Using SessionFactory sf=cfg.buildSessionFactory().
   3. SessionFactory sf=new Configuration().addAnnotatedClass(..).configure().buildSessionFactory(); In one line.
   4. Session s=sf.openSession();
   5. Under resources folder create hibernate.cfg.xml file.
5. HibernateTemplate has methods save, update, delete, get, getAll.
6. The DaoImpl will depend on HibernateTemplate which uses the SessionFactory bean.
7. LocalSessionFactoryBean implements SessionFactory. It takes(dataSource,hibernateProperties,annotatedClasses).
8. Hibernate Properties
   1. key:value pair
   2. hibernate.show\_sql=true. Shows the query
   3. hibernate.format\_sql=true
   4. hibernate.dialect=org.hibernate.dialect.MySQLDialect as SQL query changes with different databases, dialect can be mentioned.
   5. hibernate.connection.driver\_class
   6. hibernate.connection.url/username/password
   7. “hibernate.hbm2ddl.auto”>create/update</> creates a table if does not exist.
   8. cfg.addAnnotatedClass(“”) marks the entity classes.
   9. Session.save(); saves the object into table
   10. Session.persist() ensures Insert statement is executed at time of closing the transaction.
   11. Transaction t=session.beginTransaction();
   12. Without transaction any update or save on db will not work.
   13. t.commit(). Commits the transaction. For that Transaction object is created.
   14. <mapping class=”com.bean.entityname”/> is the xml alternative of addAnnotatedClass().
9. When removing a query, first get the object from db.
10. Obj=Session.get(Class,id);
11. @Entity(name=””) to create a persistent entity, storing values in the db.
12. @Table(name=””,indexes={@Index(name=””, columnList=””},uniqueConstraints={UniqueConstraint(name=””,columnNames=””)}) to change the table name, add an index to a column or an unique constraint.
13. @Id
14. @Column
15. @Transient(this field is not saved in the db, but the object has it, used for temporary calculations)
16. @Embeddable added on the class (the fields of the embeddable class are stored in the same table as the parent), in the parent class @Embedded is added to this field.
17. DriverManagerDataSource is inserted into SessionFactory. It takes properties.
    1. <property name=”hibernateProperties”><props><prop key=”hibernate.dialect>MySQLDialect</prop>
    2. <prop key=”hibernate.show\_sql”>true</prop>
    3. Properties annotatedClasses. Which takes in list of annotated classes.
18. <tx:annotation-driven/> allows transactions. @Transactional. If any error, all the changes will be reverted back otherwise committed.
19. After hibernate 6, HibernateTemplate has been removed and SessionFactory is used directly.
20. Inside hibernate.cgf.xml. This file must be present under src/main/resources. configure(“path to the file may be specified”). <hibernate-configuration> <session-factory> <property name=””></property></></>
21. Relationships:
    1. One-One
    2. One-Many
    3. Many-One
    4. OneMany and ManyOne creates intermediary table which has a composite key of the id’s from the tables. To stop the creation of this extra table and stop redundancy, we create an extra column in the table.
    5. @OneToMany(mappedBy=”order”) it tells that the other class(example Product) with the variable order has the foreign key in its table.
    6. In that entity @JoinColumn(name=”order\_id”), creates a foreign key which references the order table.
    7. Generally, One entity contains a list/ collection of multiple entities. In that entity, mappedBy used. The other entity holds a reference to a single entity where JoinColumn is used. This is done to prevent redundancy.
    8. ManyToMany
    9. @JoinTable(name=”name of the join table”, joinColumns= @JoinColumn(name=”sId”), inverseJoinColumns=@JoinColumn(name=”cId”)) is added over the List courses of one entity.
    10. @ManyToMany(mappedBy=”courses”) added over List students in entity Course; this means the Student entity is the owner of the join table.
    11. joinColumns defines the foreign key column for the owning side. inverseJoinColumns defines the foreign key column for the other side.
    12. Only one entity is the owner of the join table thus jpa knows which side to trust for maintaining the relationship. Conflicts when one side tries to insert or delete. Other side is unaware. Thus, Consistency, Performance, Data Integrity is kept.
    13. OneToMany(mappedBy,cascade=CascadeType.ALL) cascade type ensures the child class follows the parent class, any change in the parent, the associated child entry also changed. Lifecycle of child entity depends on the parent entity.
22. @OneToMany(mappedBy,cascade,orphanRemoval=true) this orphanRemoval removes orphan nodes.
    1. There is a parent class called User. A child entity called CartItem. Suppose I remove the cartItem from the user entities cart item array list. But the cart item is not removed from the database, where it sits like an orphaned row.
23. Hibernate has Cache lvl 1 already implemented. Thus, if inside a session an object persists(), and it is also fetched in that session, then hibernate does not fetch the data from the table, as it is available in the cache, saving time if the object is very big.
24. Fetchtype of hibernate is LAZY inbuilt.
    1. There is onetomany relationship within an entity. On doing select, multiple tables are joined. So, as it is lazy it will not fetch from multiple tables until the data is going to be used.
    2. @OneToMany(fetch=FetchType.EAGER)
25. Repository extends JpaRepository<ClassName,Type of Primary Key>. We can run all queries without writing them.
26. @JoinColumn(name=””,unique=true) will set a foreign key.
27. @DiscriminatorValue(“”) discriminates the subclasses when there are subclasses.
28. @DiscriminatorColumn(name=””, discriminatorType=DiscriminatorType.STRING) this is written in the parent class to create another column which will specify the value of the column.
29. @Inheritance(strategy=InheritanceType.SINGLE\_TABLE) added on parent class. Indicates all subclasses will share the same table.
30. There is Hibernate API Method and HQL.
31. HQL
    1. Session.get(Laptop.class,id) will fetch all the values of laptop. To create custom queries: use HQL
    2. Query query=session.createQuery(“from laptop”); //fetches all values
    3. Query.getResultList() to view the fetched columns from the query. Returns a list.
    4. Session.createQuery(“from laptop where ram=?1)
    5. In HQL we have to give numbers to the ?. query.setParameter(1,ram which is a variable);
    6. In HQL, we have a query with a variable written as :var, query.setParameter(“var”,variable), the variable is present in the class.
    7. getResultList() returns the entity object when all columns are selected, otherwise if more than 1 columns then Object type array returned which can be printed using for loop, and typecasting the array elements to String. Only 1 column returned in String type.
32. Session.get(className,id) fetches data eagerly. Session.load(className,id) fetches data lazily.
33. In hibernate projects, transactions need to be managed manually, while in spring boot projects with jpa transactions are managed automatically reducing boiler plate code.

Spring MVC works on servlets.

Spring JPA:

1. Spring boot starter data jpa must be added. Then it can be configured in application.properties
2. The mysql jdbc connection is done as follows:
   1. Spring.datasource.url
   2. Spring.datasource.username
   3. Spring.datasource.password
   4. Spring.datasource.driver-class-name
   5. Spring.jpa.hibernate.ddl-auto=update
   6. Spring.jpa.show-sql=true
3. Repo.save(Entity);
4. Repo.saveAll(List);
5. Repo.findAll()
6. Repo.findById() returns optional<entity obj>. optional to handle null values. Obj o=repo.findById(1); o.orElse(new Obj); will create a new obj if null else do nothing.
7. @Query(“select s from Student s where s.name=?1”) List<Student> findByName(String name); These queries are called JPQL. Here, class names and property names are used.
8. DSL(Domain Specific Language) used by JPA, and those methods are created based on the column names, and used directly. The method names start with findByColumnGreaterThan/LessThan/Equal

SLF4J

1. Logger log=LoggerFactory.getLogger(ClassName.class);
2. Log.info(adds the info to the log);
3. Log.error(saves an error);

Lombok

1. @Getter
2. @Setter
3. @ToString
4. @EqualsAndHashCode
5. @NoArgsConstructor
6. @AllArgsConstructor
7. @RequiredArgsConstructor
8. @Data is shortcut for getter,setter,toString,equalsAndHashCode,requiredArgsConstructor.
9. @Builder implements the builder pattern. User.builder().name().age()
10. @Value creates immutable classes.
11. @Setter(onMethod\_=@Autowired)

Spring REST(Representational State Transfer)

1. The RestController blends @Controller and @ResponseBody turning classes into efficient web controllers that directly handle HTTP requests and quickly return data, skipping traditional views.
2. @GetMapping(“/endpoint”) for safely retrieving data without altering server data.
3. @ComponentScan used to specify the packages that Spring should scan for components, configurations, services.
4. @PostMapping(“”)
5. @GetMapping(“{eventid}”) Event findEventById(@PathVariable(“eventId”) int eventId){}. Inside The annotation any variable, it is mapped to other variable using @PathVariable(“”). Or Event findEventById(@PathVariable int eventId);
6. @RequestMapping(“/”) used to map HTTP requests to specific controller methods.
7. @PutMapping() used to overwriting existing body.
8. @DeleteMapping()
9. @PatchMapping() for making partial updates to existing records without needing to send entire object.
10. @ResponseStatus(HttpStatus.NOT\_FOUND--404/FORBIDDEN--403/UNAUTHORIZED--401/BAD\_REQUEST--400/FOUND--302/OK--200/CREATED--201/NO\_CONTENT(204))
11. @RequestBody deserializes the JSON.
12. @RequestParam used to send parameters over request.
13. Spring managed bean is marked for a class with annotation @Component.
14. @RequestMapping(value=””) if written above a method, automatically makes it accept all request types. If accessed by browser it will default to GET.
15. @JsonIgnoreProperties used to specify properties that should be ignored during serialization and deserialization of JSON. Helps prevent unwanted fields from being included when converting objects to json and vice versa.
16. @Column(unique=true) will make a column unique.
17. @GeneratedValue(strategy=GenerationType.IDENTITY) will auto generate the id, tells the db to handle it.
18. Jackson converts object to json. Jackson Dataformat XML used to accept/send XML files. Using ResponseEntity<Product> product object can be sent using json format.
19. @GetMapping(path=””,produces={“application/json”})
20. @PostMapping(path=””,consumes={“application/xml”})
21. In a bidirectional relationship when a table is fetched, an infinite loop runs. Where one entity calls another entity continuously. Solution:
    1. @JsonIgnore: ignores user field in serialized JSON. If added to user inside cart it will not serialize.
    2. @JsonManagedReference(added in parent entity) and @JsonBackReference(added in child entity). Eliminates the loop while keeping the relationship structure.
    3. DTOs are used.

H2

1. In memory database.
2. spring.datasource.url=jdbc:h2:mem:soh
3. spring.datasource.driverClassName=org.h2.Driver

MicroServices

1. When all the services are in an app, it is monolithic architecture. Tight Coupling between the different services. All the services are packaged in a war file, so any change in one service will require redeploying the entire project, thus slowing development. Entire app is written in a tech stack. Entire app needs to be scaled.
2. Microservices give some relief to this system. It is an architecture, not a framework. A framework has tools that help rapid development of apps. Like spring boot, express for node js. This architecture follows single responsibility principle, a service has only one task to do.
3. Multiple independent services, each with their own db, each can be written in a different tech stack. This promotes scalability, as the required service can be scaled separately. If one service fails, the entire apps need not fail. Each service can have multiple instances ensuring continuity if one fails.
4. The microservices communicate in these methods:
   1. Asynchronous. The service sending the request does not wait for the sender. It continues to execute its task. It uses a message queue. Message brokers help the services to communicate with each other. A service communicates to a broker and that broker communicates to the other service. Ex: Apache Kafka, RabbitMQ.
   2. Synchronous. The sending service waits for the response from the other service. Ex: RestTemplate.
5. API Gateway is a layer between the client and microservices. It holds the ports and hostname for microservices.
6. Config Server stores the configs of the microservices, much better than each microservice having its own config and when required change them specifically, very problematic when large number of microservices present.
7. If a service goes down, other service contacting it keeps doing so, to limit the number of times it is contacted, Circuit Pattern used. It also sends a default response back to the API Gateway, incase an error occurs. Ex: Circuit Breaker Pattern.
8. Service Registry and Discovery, stores hostname and port number. API Gateway uses this. It maintains a list of the instances that are up now.
9. Distributed Tracing keeps a log of the execution hierarchy from client to microservices. Ex: Spring Cloud Sluthe.

Problem statement, solution overview, technology stack, flow diagram, key features and benefits, competitive advantage, business impact and roi, call to action.

Maximum 6 lines in a slide. Start presentation with a question, it presents the stage for the presentation.

1. Hash Password
2. RequestParams? Or PathVariable? Because user should not know what user id is being used, or even the product id. Tackled by session, jwt token.

Discount in cart,

Product registered successfully rather than return the product.