**Spring**

1. Spring works on 3 principles
   1. DI / IOC
   2. AOP
   3. Abstraction
2. Every class in spring Is called as a bean
3. Steps to create a spring Maven Project
   1. File -> new -> Other -> maven -> Maven Project
   2. Archetype : quickstart
   3. Group Id : package name
   4. Artifact Id : project name
   5. Update pom.xml file
      1. Java version : its different till jdk 8 and beyond jdk 9   
         Within <properties> tag
      2. Add dependencies
         1. Spring-context
         2. To add version at 1 place and use it using ${spring.version}  
            <properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>11</java.version>

**<spring.version>5.3.23</spring.version>**

**</properties>**

1. Create a class => **follow the naming conventions : getters/ setters**
2. Create a xml file with the name  
   spring-core.xml [ filename can be anything ]
   * 1. Configure our beans using <bean>
     2. Id attribute => instance of the class
3. Load xml file
   * 1. In main method  
        ApplicationContext context = new

ClassPathXmlApplicationContext("spring-core.xml");

* + 1. The moment the xml file is loaded all the beans configured using <bean> tag are loaded and instantiated.

1. Dependency injection
   1. Constructor
      1. XML : <constructor-arg>
      2. Annotation : **@Autowired or @Value**
   2. Setter
      1. XML : <property>
      2. Annotation : @Autowired or @Value
   3. Field injection : only via annotations : @Autowired or @Value
2. Namespaces
   1. Context namespace:  
      <context:annotation-config/> : scanning dependency injection  
      <context:component-scan basePackages =”<path to package>/> : scanning @Component(“”) and all DI annotations   
        
      By default the id generated by spring is the camel case of the classname
   2. P namespace : property
   3. C namespace : constructor
3. Scoping : specifies the no of instances per application  
   scope=””  
   @Scope
   1. Singleton : eager loading
   2. Prototype : lazy loading
   3. Request
   4. Session
4. Lazy initialization :
   1. XML : lazy-init =”true”
   2. Annotation : @lazy
5. Java based configuration : replace xml file with a class
   1. @Configuration : the class annotated with this annotation is the place where application specific configuration lives
   2. @ComponentScan : for the beans annotated with @Component and @Bean annotation
6. @Bean : used on the method level for any DI
7. Bean Lifecycle :
   1. Initialization
   2. Set properties
   3. Create bean name
   4. @PostConstruct or implement InitializingBean to intercept the bean for any preference settings before the bean is ready to be used
   5. Bean is ready : getBean()
   6. Destroyed : @preDestroy or implements DisposableBean
8. Aware Interfaces : to get the reference of the spring context and be aware of various beans  
     
   **SPRING JDBC**
9. It uses template design pattern which is a wrapper over the plain JDBC API
10. To integrate database in spring application
    1. Add spring-jdbc and the respective database driver[oracle, mysql etc… ] in pom.xml file
    2. Various implementations of DataSource interface provided by spring based on the database type
    3. DriverManagerDataSource implementation to connect woth database providing the connection parameters
    4. Created a properties file within resources folder with all the database connection parametres and used @PropertySource to provide the name of properties file
    5. For RDBMS spring provides with JdbcTemplate as a wrapper over plain JDBC APi which needs reference of DriverManagerDataSource
    6. Using @Bean the DriverManagerDataSource and JdbcTemplate is configured
    7. Autowire the JdbcTemplate in the class that needs to execute CRUDoperations
11. FOR DML : update()
12. For fetch : queryXXX()
13. RowMapper to map the database columns with the class properties
14. Transaction Management
    1. Add spring-tx in pom.xml
    2. Add @EnableTransactionManagement on the configuration class
    3. @Transactional on the method or class level

**SPRING MVC**

1. MVC : M – Model [entity class that represents mapping with database table]  
   V – View – UI [ where the user starts interacting with your application   
   C – controller [ responsible for getting the model if required, passing it to the view and return the viewname ]
2. Configure spring applications for mvc:
   1. Webapp archetype
   2. Maven dependency : servlet/ jsp / spring-web/ spring-webmvc
   3. Load the DispatcherServlet : AppConfig extends AbstractAnnotationConfigDispatcherServletInitializer
   4. Configure the view resolver and any other application specific configuration  
      @Configuration  
      @ComponentScan  
      @EnableWebMvc  
      WebConfig extends WebMvcConfigurer
   5. Create a class that is annotated with @Controller
      1. @RequestMapping
      2. @GetMapping
      3. @PostMapping
3. By default all the hits to the server is a GET request
4. POST, PUT or DELETE requests needs to be explicitly configured
5. POST => <form>
6. REST API : PUT or DELETE
7. Flow of spring MVC application
   1. Request comes to DispatcherServlet [DS ]
   2. From DS to BeanNameUrlHandlerMapping [ BNUHM ] that looks up for java classed annotated with @Controller annotation and any method mapped for the respective path in the url  
      <http://localhost:8080/SpringMVCDemo/greet>  
      Looks up for method with @RequestMapping(“greet”)
   3. BNUHM returns the viewname + model to DS
   4. From DS to ViewResolver where the path of the view using suffix and the extension using prefix is configured
   5. From VR to DS to the User

**SPRING AOP**

1. Terminologies
   1. Aspect -> common concern or common functionality across your application like : logging, transaction, security, cache etc..
   2. Advice : A piece of information of when to apply a particular business logic
      1. Before
      2. After
      3. AfterReturning
      4. AfterThrowing
      5. Around
   3. Pointcut expression : Ann expression that tells where to apply a particular advice
   4. JoinPoint : The point at which the advice is actually applied at runtime to get the information about the target object and the method
   5. Weaving : is a concept of applying an advice around the target object
   6. Proxy : A proxy class of the target object is created to weave in the advice
2. Annotations:
   1. @EnableAspectJAutoProxy : to enable AOP
   2. @Aspect : the class that provides with common concerns
   3. @Before
   4. @After
   5. @Aroung
   6. @AfterReturning
   7. @AfterThrowing
   8. @Pointcut
3. AOP is apoplied only at runtime and not at the time of the spring context being created

**SPRING BOOT**

1. Provides with production ready application with less configuration
2. It is an opinionated framework that provides with configurations based on the build path configured
3. Steps to create a spring boot application
   1. Either use start.spring.io or create a Spring starter project from STS
   2. Configure:
      1. Maven or gradle
      2. language
      3. Java version or version of your chosen language
      4. Group id
      5. Artifact id
      6. Packaging type : jar or war
      7. Add dependencies :
         1. Spring-data-jdbc
         2. Spring-data-jpa => ORM and Hibernate
         3. Database driver
         4. lombok
      8. Configure in application.properties the database connection paramters
4. Spring boot provides with XXXRepository to perform CRUD operations
   1. CrudRepository : all the crud operations
   2. JpaRepository
   3. PagingAndSortingRepository
5. Hibernate annotations
   1. @Enitty => specifies the class is database managed entity. Hibernate automatically looks for a table with the class name and maps the class properties with the table columns
   2. @Table : if database table name and class name are different [optional ]
   3. @Id – identifying which property is a primary key
   4. @Column [optional ] : if column name and property names are different. Can csutomize the column properties Ex: length or nullable, unique
   5. @OneToOne -> one to one mapping
   6. @OneToMany
   7. @ManyToOne
   8. @ManyToMany