**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

**SPRING BOOT DATA JPA**

1. Spring boot provides with XXXRepository to perform CRUD operations
   1. CrudRepository : all the crud operations and is at the base
   2. PagingAndSortingRepository : filtering and sorting the data
   3. JpaRepository : provides more features along with above 2 interfaces
2. 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 customize the column properties Ex: length or nullable, unique
      1. Length
      2. Name
      3. Nullable
      4. columnDefinition
      5. unique
   5. @OneToOne -> one to one mapping
   6. @OneToMany -> (mappedBy=”<property name of another class with which it is mapped) . A 3rd table is not created
   7. @ManyToOne
   8. @ManyToMany
   9. Uni-directional and Bi-directional
   10. @JoinColumn(name=””, referencedColumn=””)
   11. @JoinColumns(@JoinColumn(),..)
   12. @GeneratedValue -> for auto-increment or strategy to generate values for column
   13. Composite keys in hibernate

**SPRING BOOT XXXRepository**

1. Interface extends CrudRepository<Type, Id>
   1. It provides with all the basic CRUD operations
      1. Insert
      2. Update
      3. Delete all
      4. Delete by id
      5. Select by id
      6. Select all
      7. Count
2. Default Queries methods **BUT FOLLOW THE CONVENTION**  
   findByPropertname or getBy  
   <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods>
3. Custom Queries  
   @Query -> native query, JPQL -> Select
4. @Modifying -> DML

**SPRING BOOT MVC**

1. Add spring-web dependency => DispatcherSrvlet is configured automatically
2. For JSP -> tomcat-embed-jasper [MVC ]
3. ViewResolver in application.properties file : prefix and suffix
4. Other MVC functionality remains the same as in spring MVC

**SPRING BOOT REST**

1. REST -> Representational State transfer, it represents the state of data to be transferred over the network
2. REST is protocol independent but leverages HTTP for communication over internet i.e web services using REST API
   1. HTTP Methods : GET POST PUT DELETE PATCH ..
   2. HTTP STATUS CODES : 1XX, 2XX, 3XX, 4XX, 5XX
   3. HTTP HEADER : Accept, Content-Type, Charset, length, Authorization
3. Convention to create URI – Uniform Resource Identifier
   1. URI should be nouns with s for plural  
      http://localhost:8080/product/brand/{brandname}  
      http://localhost:8080/product/ {brandname}
   2. http://localhost:8080/customers/customername  
      @PathVariable

<http://localhost:8080/product?brand=brandname>  
@RequestParam

* 1. verbs => Http methods
  2. HATEOS -> Hypermedia as the engine of application state
     1. Reference links to other rest API

1. How to configure REST in spring boot
   1. Spring-web in pom.xml
   2. Db related dependencies
   3. Create a class
      1. @RestController [ @Controller + @ResponseBody ]
      2. @GetMapping(produces)
      3. @PostMapping(consumes)
      4. @PutMapping
      5. @DeleteMapping
      6. @PathVariable
      7. @ResponseStatus
      8. @RequestBody -> to read the payload sent iin the request body by the client application
      9. ResponseEntity -> to wrap the response with other information like status etc..
      10. @RequestParam – For query parameters
   4. Exception Handling
      1. @ExceptionHandler ->add on a method to handle for a certain type of exception
      2. @ControllerAdvice
   5. Spring – data – rest
      1. @RepositoryRestResource ->for CRUD operations and expose the data over REST endpoints

**SPRING BOOT LOGIN AND INTEGRATE WITH ANGULAR**

1. Created a PostMapping(“/login”) -> LoginUser
   1. Returned success for valid
   2. Returned failure for invalid
2. From angular make a post request from the login page -> and for success, store the email in the localstorage and redirect to dashboard. For failure, toggle the loginfailed and stay on the login page with error message displayed
3. CORS -> Cross origin request sharing which doesn’t allows cross origin communication. Then enable cors on the server side applications.

**SPRING BOOT ACTUATOR**

1. Actuator is a application monitoring and tracing tool
2. Add spring-boot-starter-actuator dependency
3. Exposes 3 endpoints on <http://localhost:8080/actuatot>
4. To enable other endpoints  
   For specific endpoints  
   management.endpoints.web.exposure.include=emp, health, info  
   For all endpoints   
   management.endpoints.web.exposure.include=\*
5. Change the actuator base-path  
   management.endpoints.web.base-path=/manage
6. Change the actuator port  
   management.server.port=8081
7. Can add more functionalities to the existing endpoints by creating a class that extends that endpoint : HealthIndicator
8. Create custom endpoint as follows:
   1. Add @Endpoint and @Component
   2. Create a method with @ReadOperation

**SPRING BOOT SWAGGER**

1. Swagger is used for documenting REST API, to test and check for the types of inputs and outputs
2. Add spring-doc-openapi dependency
3. Exposes the swagger ui on  
   <http://localhost:8080/swagger-ui.html>
4. Customize the header by creating an object of OpenApi()
5. Add meta data to schemas using various different annotations
6. Add meta data to rest controllers using @ApiResponse

**Java Logger**

1. It has 3 main components
   1. Logger => that will record logs for that specified class
   2. Appenders or hanldlers => handle the logs and send it to the respective destination
   3. Layout or formatter -?> that specifies the pattern of the generated logs
2. FileHandler – default is XML can override by creating a custom formatter
3. ConsoleHandler

**Spring Security**

1. Authentication : who logged in i.e the principal
   1. LDAP
   2. Credentials based
   3. Jwt
   4. Oauth etc…
2. Authorization : what the authenticated user can access
3. Principal
4. Roles – group of permoissions
5. Granted Authority
6. Steps to implement spring boot security
   1. Add spring security dependency -> all urls are by default protected and username is user and password is auto-generated in the console
   2. To override the default credemntials
      1. Add username and password in application.properties
      2. Create a security configuration class that extends WebSecurityConfigurerAdapter and add foll annotations
         1. @Configuration
         2. @EnableWebSecurity
      3. Override configure(AuthenticationManagerBuilder) method
         1. In memory authentication
         2. Database
7. Spring provides encapsulation over the different providers of the data to be validated for success or failure login
   1. Implement the UserDetailsService -> loadByUsername(String username) -> this username is the name entered by the user in the login form and it returns UserDetails interface for which User is the implementation class. User is from spring security  
      class MyUser implements UserDetails   
      Using the username fetch for the user data from the respective authentication provider (CrudRepo -> H2, SQL or no-SQl, LDAP)
   2. Autowire the service class in security config class and provide the reference to AuthenticationManagerBuilder -> auth.userDetailsService(yourservice);
8. Authorization:
   1. Override the configure(HttpSecurity http) -> that provides with restricting role based access to urls and which ones to permit
9. CSRF -> Cross site request forgery => is a token that is added by default by the spring security
   1. Name=\_csrf -> unauthorized access or any redirect of the urls
   2. http.csrf().csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse()); -> configure(HttpSecurity)
   3. REST API with JWt or any other token based security mechanism -> disable csrf -> rest api stateless by default
   4. SecurityContextHolder

**Spring Security - JWt**

1. JSON web token
   1. Header
   2. Payload
   3. signature
2. Add json-web token dependency
3. JwtUtility -> generate / validate the token
4. Filter extends OncePerRequestFilter -> we added this filter before all the filters within security configuration file or class using http.addFilterBefore()
5. Controller : /auth [ POST ] and it return the jwt token : key-value : token:

**Spring Security - OAuth**

1. Oauth – open API authentication
2. It is used for 3rd party authentication
3. Terminologies
   1. Resource Owner
   2. Resource server
   3. 3rd party client
   4. Authorization server ->
      1. Client – register their application and get the client credentials generated by the authorization server   
         client id, secret key, expiry, scope,
      2. Once the client is registered , they can use the Google credentials to login to the client portal
      3. Once you click on google link on the client website , you are redirected to google login page and the resource owner needs to authenticate
      4. Once authentication successful, redirected to the approval with the set of permission the client needs to access on your behalf
      5. If approve -> then a code is sent by the authorization server to the client
      6. Using the code the client needs to authenticate itself and get the access token
      7. Using the access token , client can access the resources from the resource server
4. Add oauth dependency
5. @EnableAuthorizationServer
6. @EnableResourceServer
7. Normal spring security to authenticate and authorize the urls

**Microservices**

1. go for microservices when
   1. scalable - horizontally
   2. flexible
   3. make it available 24/7
   4. business is defined
   5. create small programs
2. Create microservice
   1. Server
      1. Eureka server – discoverable, monitors the health and provide with load balancers
      2. @EnableEurekaServer
      3. Modify properties file
      4. Default port is 8761
   2. Client –
      1. eureka client that helps 2 services deployed on the same server or registered on the same server communicate with each other.
      2. Automatically registers themselves on the eureka server running at port 8761. If the eureka server port is changed configure within the eureka clients  
         eureka.client.serviceUrl.defaultZone=http://localhost:8082/eureka/
   3. 2 microservices communicate over RESt API using RestTemplate object
   4. Create @Bean method that returns RestTemplate object and add @LoadBalanced for load balancers to balance the load between different instances of the same application
   5. Give a unique id to you services by using:  
      spring.application.name
3. Feign Clients –
   1. a wrapper over RestTemplate API
   2. just specify what and how wil be taken care by feign
   3. Add feign dependency
   4. Annotate main class with @EnableFeignCLients
   5. Create an interface with annotation @FeignClient provide the location of the service – hardcoded url or service name
   6. No need of load balancers as feign takes care internally
4. Never return a fixed type form a method but do return a DTO
5. Design Patterns
   1. Circuit breaker -> for any exception thrown by amy microservice – hystrix[deprecated or resilience]
   2. Gateway – a filter on the front of all the microservices  
      Zuul or spring-cloud-api-gateway  
      add security and logging etc any common functionalities across microservices.  
      It provides with a single entry point to your application
   3. Externalize configuration – cloud config
   4. Monitoring and tracing tools - zipkin

**Reactive Programming**

1. It is a programming model amongst others like imperative, declarative, functional, OOPS, event driven etc..
2. It focuses on change of data
3. Used specifically where continuous live stream of data needs to be processed continuously
4. Implementation of reactive paradigm
   1. Rxjs
   2. Project-reactor
5. Terms
   1. Publishers
      1. Mono – 0 or 1
      2. Flux – 0 or more
   2. Subscriber
   3. Subscription
   4. Processor
   5. Backpressure
6. A simple maven quickstart project -> with project-reactor as a dependency
7. Various operators to be used with Mono and Flux
8. Spring boot web project with webflux as a dependency for reactive programming
   1. Netty is the default server
   2. MediaType => Stream\_event

**Spring Batch**

1. It is a programming model that is used to execute tasks in the background
   1. Migration of data from 1 db to another
   2. Report generation
   3. Billing
2. User intervention is not there
3. Terminologies
   1. Trigger -> JobLauncher
      1. Job
      2. JobParameters
   2. Job
   3. Steps
      1. Chunks – long lasting
         1. ItemReader
         2. Process
         3. ItemWriter
      2. Task oriented – short lived
4. To create a spring boot project
   1. Web, database related, spring-batch:dependencies
   2. Properties : db, batch.enabled=false, schema=ALWAYS
   3. Configuration : @EnableBatchProcessing and @Configuration
      1. JobBuilderFactory
      2. StepBuilderFactory
      3. Reader - lineMapper
      4. Writer
      5. Processor
      6. Steps
      7. Job
      8. TaskExecutor -> parallel
   4. Launch the job by calling the run() of JobLauncher
   5. Listeners -> Job, Step , Reader, Writer, Processor
   6. Partitioning -> spawn more thread based on the number of cores your CPU has
   7. Fault Tolerance -> step, job, reader, writer