**Spring Boot**

**Spring boot is equal to spring framework without xml configuration and with integrated server.**

**Why springboot?**

* **We don’t need xml based configuration**
* **Development is very fast, bcz it has inbuilt tomcat/jetty server**
* **we can simply create spring boot application by using spring initializer and by using enable auto configuration we can avoid boiler plate code**

**Create Spring boot application:**

1. **From scratch by adding dependencies manually**
2. **Using spring initializer**
3. **Using STS - > file -> new -> spring starter project will create a new project.**
4. **Using Spring boot CLI**

**Spring initializer:**

**Allows to create spring boot applications quickly. we don't need to create from scratch.**

[**https://start.spring.io/**](https://start.spring.io/)

**It will allow to select the dependencies as a starters which will add respective jars to the classpath**

[**https://www.baeldung.com/spring-boot-gradle-plugin**](https://www.baeldung.com/spring-boot-gradle-plugin) **-- create spring boot application from scratch**

* **When we run application in eclipse it will create binary class files and deploy those class files into embedded tomcat**
* **When we run application using command prompt - gradlew clean build it will generate the executable jar file in build\libs folder**
* **We can run this jar by using java -jar “jar-name”.jar**
* **To create war build add apply plugin: 'war' in build.gradle and run gradlew clean build it will generate the executable war file in build\libs folder**

**STS**

* **STS - spring Tool Suite**
* **Sts is extended IDE of eclipse**
* **It is specialized tool to create spring boot application**
* **We can add STS add plugin in eclipse**
* **Click help-> eclipse marketplace**
* **Search with STS and install sts plugin**
* **Create new project by clicking file->new->spring starter project**

**Spring boot CLI**

* **Command Line Interface is used to quickly develop spring boot application w/o much boilerplate code**
* **It is the software to run and test the spring boot application from cmd**
* **when we run the spring boot application using CLI it internally uses the spring boot starter and spring boot auto configuration components to resolve all the dependencies**
* **Download cli here** [**https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html**](https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html)
* **Unzip it, set path verify the version by cmd spring --version**

**Disadvantage of SpringBoot:**

1. **Migration Efforts: migration from existing spring project to spring boot is not straight forward. spring boot is mostly for new development projects.**
2. **Deployment to webSphere/WebLogic/Other Appservers are also not very simple. You need to make a few changes like downgrading the JPA version, removing conflict jars etc in order to make it work in Application Servers.**
3. **Microservices & Cloud Native: Spring Boot has been developed keeping Micro services & cloud Native in mind, you may not see improvements in other areas**

**Embedded DataBase:**

1. **H2, Derby those are embedded databases**
2. **It is in-memory databases**
3. **Recommended only for testing/poc purpose**

**Advantage**

1. **It is lightweight, fast and less configuration developer can focus the development instead of configuration of data source**

**Disadvantage**

1. **We can’t use these in prod env**
2. **Because in prod we must save our data, but embedded databases life is until server up once it is restarted all data gone.**

**Spring Boot Data JPA Repositories**

**The goal of spring boot data jpa repository is to reduce the boiler plate code require for data access layers.**

1. **Spring data JPA**

* **it is part of the spring data family, aims to provide the jpa based repositories that aims to simplify the implementation of data access layer using JPA.**

1. **Spring data Repositories**

* **Spring data provides the abstract repositories, that are implemented at run time by the spring container and perform the CRUD operation.**
* **AS a developer we have to just provide the abstract methods in the interfaces.**
* **This is reduce the amount of boiler plate code required to write data access layers.**

1. **Base interfaces in spring data commons project**
2. **Repository**

* **It is the central interface in the spring data repository abstraction**
* **This is the super interface for curd repository**
* **This is a marker interface. hence it will not have any default method**
* **if we are extending this interface, we have to declare our own methods and the implementations will be provided by the spring container at run - time.**
* **For this interface we have to pass 2 parameters:1. Type of the entity 2. Id of the entity**

1. **CrudRepository**

* **CrudRepository provides methods for the CRUD operations.**
* **This interface extends the repository interface**
* **When you define crud repository, you have to pass the two parameters**
* **1. Types of the entity, 2 id of the entity**

**Methods :**

1. **S save(s entity)**
2. **Iterable<S> save(Iterable<S> entities);**
3. **T findOne(ID primaryKey)**
4. **Boolean exist(ID primaryKey)**
5. **Iterable findAll()**
6. **Iterable<T> findAll(iterable<ID> ids)**
7. **Long count()**
8. **delete(ID id)**
9. **void delete(T entity)**
10. **delete(Iterable<? Extends T> entities)**
11. **deleteAll();**

* **In crud repository we can keep blank, no need for implementing own methods for default operations.**

**JpaRepository**

* **Which is the special version specific to the jpa technology**
* **If we are not using jpa specify thing then use crud repository.**
* **Because it will not tie your application with any specific store implementation**

**3. PagingAndSortingRepository**

* **This is extension of CrudRepository**
* **It is a specialized version of the paging operations.**

**Public interface PagingAndSortingRepository<T,Id extends Serializable> extends CrudRepository<T,ID>{**

**Iterable<T> findAll(Sort sort);**

**Page<T> findAll(Pageable pageable);**

**}**

**Both JpaRepository and CrudRepository are base interfaces in Spring Data. Application developers have to choose any of the base interfaces for your own repository. It has 2 purpose**

1. **To allow spring data to create a proxy instance for your repository interface.**
2. **To inherit the maximum default functionality from the base interfaces without declaring your own methods.**

**Creating Custom Queries with query methods.**

1. **Creating Database queries from method names: describes how we can create database queries from the method names of our query methods.**
2. **Creating Database queries with Named Queries: describes how we can create database queries by using named queries.**
3. **Creating Database Queries with the @query annotation: describes how we can create database queries by annotating our query methods with the @Query annotation.**

**1.Creating Database queries from method names**

* **Spring data jpa has a built in query mechanism which can be used for parsing queries straight from the method name of a query method.**
* **This mechanism first removes common prefixes from the method name and parses the constraints of the query from the rest of the method name.**
* **The query builder mechanism is described with more details in Defining Query methods subsection of Spring Data JPA reference documentation**
* **Using this approach is very simple. All you have to do is to ensure that the method names of your repository interface are created by combining the property names of an entity object and the supported keywords.**

**Eg1: interface PersonRepository extends Repository<Person, Long>{**

**List<Person> findByLastname(String lastname);**

**}**

* **Here Lastname is a property in person entity**
* **Spring data JPA will fire the query**

**Select \* from Person where Lastname=:lastname**

* **Find- Select**
* **By - splitter , based on property**
* **Lastname - used in where condition**

**Eg1: interface PersonRepository extends Repository<Person, Long>{**

**List<Person> findByFirstNameAndEmail(String firstName,String email);**

**}**

* **Query - select \* from Person where firstName = :firstName and email=:email**
* **And in method name is keyword here to indicate operation in where condition**

**2. Creating Database queries with Named Queries**

* **We can use either named-query XML elements or @NamedQuery annotation before entity class to create named queries with the JPA query language.**
* **You can use either named-native-query XML element or @NamedNativeQuery annotation before entity class to create queries with SQL if you are ready to tie your application with a specific database platform.**
* **The only thing we have to do to use the created named queries is to name the query method of your repository interface to match with the name of your named query.**

**Eg: @NamedQuery(name=”Person.findByName”,query=”Select p from Person p where p.lastname=?1”)**

**Drawback:**

* **It is valid if our application is small or if we must use native queries.**
* **If our application has a lot of custom queries, this approach will litter the code of your entity class with query declarations(we can se the xml configuration to avoid this but it is more horrible.)**

**3.Creating DataBase Queries with the @Query annotation:**

* **The @Query annotation can be used to create queries by using the JPA query language and to bind these queries directly to the methods of your repository interface.**
* **When the query method is called, spring data jpa will execute the query specified by the @Query annotation (if there is a collision between the @Query annotation and the named queries, the query specified by using @Query annotation will be executed)**

**Eg: @Query(”Select p from Person p where p.lastname=:lastName”)**

**Public List<Person> find(@Param(“lastName”) String lastName)**

**When we are doing update, insert, delete operation we need to use @Modifying annotation along with @Query .so that spring will trigger executeUpdate().**

**Spring boot Devtools**

* **If we add spring boot devtools starter in our build tools, if we do any changes in project we don't need to restart the server.**
* **Spring Boot DevTools pick up the changes and restart the application.**

**Spring Actuator**

* **Spring boot actuator is sub project of spring boot**
* **It provides several protection web services to your application**
* **It has around 16 endpoints. to use this add spring-boot-starter-actuator dependency in build tool.**
* **It is production ready features, we can monitor our production application like how many beans are created, env info , health of application etc.**
* **once our application configured with actuator we can monitor/manage our application with different actuator http endpoint.**
* **Apart from health and info other endpoints are sensitive.**
* **Hence we need to authorize to access these endpoints**
* **To disable security we can add spring.management.security.enabled=false in application.proprties file.**
* **we can add management.port=9090 in application.proprties file to access actuator endpoints.**
* **We can customize actuator endpoints name. Eg: we can change env endpoint to myenv by adding endpoints.env.id=myenv in application.proprties file**
* **endpoints.env.enabled=false means we cant access this endpoint.so t should be true.**

**Spring Boot Security**

* **Spring-boot-starter-security dependency will add the dependencies required for spring security.**
* **Once we added this dependency by default spring boot provide inbuild form based authentication**
* **Whereas user name is user and password is randomly generated password. Password found in console.**
* **Most of the time we won’t rely on default user/password.**
* **We can provide own user name and password through application.properties file by adding security.basic.enabled=true, security.user.name=admin, security.user.password=root**

**JWT: - need to implement**

* **JWT stands for JSON web token, it provides us an way to authorize API request using token based authentication**
* **JWT token consist of 3 parts, header, payroll, signature**
* **JWT is helpful when we having lot of microservice and all services needs to authenticate**

**Common terminology:**

* **Authentication** refers to the process of verifying the identity of a user, based on provided credentials. A common example is entering a username and a password when you log in to a website. You can think of it as an answer to the question *Who are you?*.
* **Authorization** refers to the process of determining if a user has proper permission to perform a particular action or read particular data, assuming that the user is successfully authenticated. You can think of it as an answer to the question *Can a user do/read this?*.
* **Principle** refers to the currently authenticated user.
* **Granted authority** refers to the permission of the authenticated user.
* **Role** refers to a group of permissions of the authenticated user.

**Steps to implement:**

* **Add Spring-Boot-starter-security in build tool.**
* **Create class with annotation @Configuration, @ EnableWebSecurity and extend WebSecurityConfigurerAdapter**
* **Override configure method and add** authorizeRequests urls.

**Vidoe ref : https://www.youtube.com/watch?v=lA18U8dGKF8**

**Oauth – implementation**

**Spring Boot Logging Configuration**

* **Logging is useful in debugging applications**
* **Logging libraries like Logback,log4j2 provide synchronous logging which can impact application performance.**
* **In synchronous logging, the data is directly written to the destination i.e a file or a database.**
* **In Asynchronous logging, the data is written to a queue first and then to the destination.**
* **By default spring boot providing spring-boot-starter-logging which does not require any additional configuration. (used in our proj)**
* **This includes slf4j, logback dependencies**

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

**/\*\* The log object used to logging the message \*/**

**private static final Logger LOGGER = LoggerFactory**

**.getLogger(PrecautionsController.class);**

**LOGGER.info("Updating the precautions for request model {}",**

**requestModel);**

* **The visibility of log/logging level is Fatal,Error,Warn,Info,Debug,Trace,All**
* **After fatal application mostly stopped**
* **Error is used to error/exception. If we set the alert for this type Used to monitor the exception on project**
* **Warn used to log warnings eg:connection lost b/w client and server, data base connect lost**
* **Info - eg: server starting messaging, inout , output etc**
* **Trace-**
* **All- to turn on all logging level**
* **Default logging level is info. That time debug and trace will not display**
* **By default when you run the application debug logs are not displayed. To enable it in application.properties file add logging.level.root=Debug**
* **Now it will show lot of logs. hence it is recommended for dev mode. Not prod mode.**
* **To set the log pattern in console we can add logging.pattern.console= %d(yyyy-MM-dd HH:mm:ss) - %msg%n**
* **To write a logs in file add command -**

**logging.pattern.file=%d(yyyy-MM-dd HH:mm:ss) [%thread] %-5level %logger{36} - %msg%n**

**logging.path=logs - to create folder in our proj and create spring.logfile**

**Logging.file = MyappLogs/myapp.log- to create custom file in project**

* **By using logback xml configuration we can override default logging configuration**

**Embedded tomcat**

* **As part of spring boot starter web tomcat server dependencies will be installed**
* **To exclude this we need to exclude tomcat in build tool configuration**
* **Add jetty starter in dependencies**
* **If we have not add spring boot starter web application will be deployed in build Netty server , no config required**

**Eg: https://howtodoinjava.com/spring-boot/configure-jetty-server/**

**SpringBoot Profiles:**

**Spring boot profiles**

* **It is the way to segregate application configuration**
* **It makes useful in configuring different environment like dev,prod,qa**
* **In application.properties file add spring.profile.active=dev**
* **Or else we can set the profile in command prompt**
* **We can create profile under resources , eg : for dev env profile is application-dev.properties**
* **Also we can create profile using java like below**

1. **Create java interface which has all the property keys**
2. **Create dev/qa/prod/ profile classes which implements EnvBasedConfig, override parent interface method and provide individual values for each profile class. (means we are creating many beans . which will active only at the time of this profile is active)**
3. **Annotate the class with @Service, @Profile(“dev”)**

**Default port number:**

**We can change default port number 8080 using below ways.**

1. **server.port=8081 in application.properties**
2. **Server.port as key , 8081 as value in environment variables**
3. **Change using command prompt,**

**java -jar “application.jar” --server.port=8081**

**Thymeleaf**

* **Thymeleaf is a server-side template engine that can process XML, HTML etc.**
* **Thymeleaf can access class fields,message properties from i18n messages files.**
* **We can bind our class fields with HTML from elements using thymeleaf.**
* **We can iterate our java collections using thymeleaf.**
* **We can perform from validation and display i18n messages using Thymeleaf, We can also use CSS files with our Thymeleaf View.**

**Integrate Thymeleaf with Spring Boot**

* **To integrate Thymeleaf with Sping boot, we need to use following Spring boot starter in our pom.xml**

**<dependency>**

**<groupid>org.springframework.boot</groupId>**

**<artifactId>Spring-boot-started-thymeleaf</artifactId>**

**</dependency>**

* **If we want to use higher version of Thymeleaf then we need to configure following properties in pom.xml**

**1.thymeleaf.version**

**2.thymeleaf-layout-dialect.version**

**Suppose we are using Maven then the above properties will be configured as follows.**

**<properties>**

**<thymeleaf.version> 3.0.6.RELEASE</thymeleaf.version>**

**<thymeleaf-layout-dialect.version>2.2.2</thymeleaf-layout-dialect.version></properties>**

* **Now Spring Boot will use thymeleaf version 3.0.6.Release and Thymeleaf layout dialect version 2.2.2**
* **If spring boot scans thymeleaf library in its classpath then we are ready to work with Thymeleaf Spring Boot provides properties for thymeleaf that will be configured in application.properties or application.yml to change the configuration of thymeleaf with Spring Boot. we are listing some of them here.**

**spring .thymeleaf.mode : Template mode that will be applied on templates. Default is HTML 5.**

**Spring.thymeleaf.prefix: This is the value that will bd prepended with view name to build the URL. Default value is classpath/templates/.**

**Spring.thymeleaf.suffix this is the value that will be appended with view name to build the URL. Default value is html.**

**With the default Spring Boot and Thymeleaf configuration we can keep our thymeleaf files with html extension at the following location.**

**src\main\resources\templates**

**Spring Batch**

**Spring Transaction**