* Java EE – contains JSF,JSP,EJB,JMS,RMI,JPA,JSF and JAXB,JAX-WS , web sockets
* Java EE is named as Jakarta from version 8 and that community is managing Java EE from then
* Spring core – IOC, Dependency injection, beans, context, SPEL,IOC container

Pros – loose coupling, autowire, unit-testing is made easy,system maintenance and modularity, concurrent development

* 2 types of IOC – beanfactory and applicationContext
* @Bean tells spring that it needs to call this method, when it initializes its context and adds returned object/value to spring context/IOC container
* @PostConstruct is used to define our custom logic,once the bean is created (using @component etc..,)
* @PreDestroy used on top of methods,before clearing context for clearing IO resources, DB connections etc
* Register bean is used from Spring 5 to programtcly add beans

Context.registerBean(“vw”,Vehicle.class,vwsupplier);

* Wiring types

@Bean

Public Vehicle vehicle()  
{

Vehicle v = new Vehicle();

v.setVehicle(vehicle());

}

* @Bean

Public Vehicle vehicle(Person person)  
{

Vehicle v = new Vehicle();

v.setPerson(person);

}

* Public class Person(){

@Autowired – it can be done on field, setter and constructor also

Private Vehicle vehicle, In case of single constructor @Autowired is optional

* For constructor , Autowiring can be done by name ,@primary and @qualifier
* Scopes – singleton,prototype,request,session and application
* @component

@scope(BeanDefinition.SINGLETON)

* Eager is default one and we can make it lazy if not needed
* If parent bean scope is singleton, then the dependent will also be singleton, but not prototype
* Servlet container->Dispatcher servlet->
* @SpringBootApplication has 3 - @AutoConfiguration,@SBConfiguration and @ComponentScan
* #server.port = 0 – tells SB to start application at any random port
* @lombok – getter,setter,ReqArgs,toString,Equals and hashcode
* @slf4j – replaces the Logger log = LoggerFactory. getLogger(Student.class)
* @RequestParam takes name,value,defaultValue and Required as parameters

String displayHolidays(@RequestParam(required = false) boolean festival,

@RequestParam(required=true) boolean federal)

localhost:8080/holidays?festival=true&federal=true

* Spring-boot-starter-validation maven dependency for bean validations’
* @valid and Jakarta.validation.constraints.\* and hibernate.validator.constraints.\*
* Singleton,request,prototupe.session and application
* Session scope is at user-level, app scope is for whole app and request scope is at request level
* Spring-boot-starter-security is the default security dependency
* <https://github.com/eazybytes/spring>
* Spring.security.user.name
* Spring.security.user.password
* Starting from **Spring Security 6.1** and **Spring Boot 3.1.0**versions, the Spring Security framework team recommends using the Lambda DSL style for configuring security for APIs, web paths, etc. Consequently, they have deprecated a few methods within the framework. These deprecated methods are planned to be removed in **Spring Security 7**, which is expected to be released in the next 2-3 years. This timeframe allows all developers sufficient time for migrating their code.
* permitAll() allows any request to use webapp
* denyAll() – to deny all

Eg: @Configuration

Public class ProjectSecurityConfig{

@Bean

SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception{

http.authorizeHttpRequests().anyRequest().denyAll().and().formLogin().and().httpBasic();

return http.build();

* http.authorizeHttpRequests().requestMatchers(“/home”).permitAll();

http.authorizeHttpRequests().requestMatchers(“/about”).permitAll();

http.csrf().disable().authorizeHttpRequests()

* @Bean

Public InMemoryUserDetailsManager userDetailsService(){

UserDetails admin = User.withDefaultPasswordEncoder()

.username(“user”)

.password(“pwd”)

.roles(“USER”)

.build();

return new InMemoryUserDetailsManager(user,admin);

* http.authorizeHttpRequests().requestMatchers(“/about”).permitAll()

.and().formLogin().loginPage(“/login”)

.defaultSuccessful(“/dashboard”).failureUrl(“/login?error=true”).permitAll()

.and().logout().logoutSuccessUrl(“/login?logout=true”).invalidHttpSession(true).permitAll().and().httpBasic();

* @ControllerAdvice – handles all global exceptions in one component,it acts like an interceptor of exceptions thrown by @RequestMapping
* To prevent CSRF attack,a secure CSRF token need to be generated for each user session
* CSRF token is used by app to verify legitimacy of server and request is coming from same UI or not, app server rejects request if CSRF token didn’t match
* By default, Spring security enables CSRF for all http methods where data-excahnegs happens except for GET, thymeleaf also has good support for CSRF

<input type=”hidden” th:name = “${\_csrf.parameterName}” th:value=”${\_csrf.token”}

* CSRF is token in hidden value in UI side and cached on server side

CSRF can be disabled at whole app level or at any specific request level ,as below

http.csrf().disable()

http.csrf().ignoringRequestMatchers(“/saveMsg”)

Eg:

@Bean

SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws Exception{

http.csrf().ignoreRequestMatchers(“/saveMsg”).and()

.authorizeHttpRequests()

* JDBC STEPS

Load driver class

Obtain db connection

Prepare sql statement using connection object

Execute query

Process result set

Close connection

* Spring-jdbc has all predefined templates

JDBCTemplate

NamedParameterTemplate

* @Bean

Public DataSource myDataSource(){

Dribermanagersoruce dms = new DriverManagerSource();

dms.setDriverClassName(“jdbc.driver”);

dms.setUsrname();

dms.setPassword();

return dms;

}

We can use JDBCTemplate within DAO implementation with datasource ref or configure it in IOC container and give it to DAO as bean reference

Int count = this.jdbcTemplate.queryForObject(“select \* from person wherr name = ?”,Integer.class,”Joe”);

* Inside any Repo/DAO where we want to execute queries,we need to create bean/object of JdbcTemplate by injecting source bean

@Repository

Public class PersonDAOImpl implements PersonDAO{

JdbcTemplate jdbc;

@Autowired

Public PersonDAOImpl(DataSource datasource){

jdbcTemplate = new JdbcTemplate(datasource);

}

jdbcTemplate are thread-safe, so we can inject shared ref to multiple DAO’s

* Return jdbctemplate.update(sql,contact.getName(),contact.getMobile())
* IF COLMN NAMES AND field names in pojo/bean are same, then we can use the

BeanPropertyRowMapper given by spring

String sql = select \* from employees;

Var rowMapper = BeanPropertyRowMapper.newInstance(Holiday.class);

return jdbcTemplate.query(sql, rowMapper);

* SB autoconfigures datasource,jdbctemplate and NamedTemplate based on DB connection details in property file and we can @Autowire them directly into our own repository classes.

Spring.jdbc.template.max-rows = 500;

**Spring-Security**

* Spring gives 2 types of tokens i.e. JSESSIONID and XSRF-TOKEN, stored as cookies inside browser, any request from UI to BE, these 2 tokens will be attached to the backend and spring-security framework will attach these tokens,

Because of these tokens, UID and pwd need to be passed everytime and backend also need not perform authentication for each and every request,

* XSRF token protects us from CSRF attacks
* Opaque tokens are random simple strings maintained by backend server, to check and there is always dependency on BE server to validate each and every request,
* JWT tokens are not dependent on auth-server and can do self-validation,where as opaque tokens are used in case of auth-servers
* JWT can be self-validated using token sign wth public key,without auth server call
* Security,reusability,multiple-platform,self-contained,statelessness and expiration.

JWT Eg: Hmac256(base64encode(header)+”.”+base64encod(payload,secret);

Secret is know to backend server only.

**OAUth2:**

Example is end user will let google photos issue temp access token, using this temp access token can only read images from app

* OAUth is open-authorization built on IETF standards,OAuth2 is built on top of OAuth and security standard where one app gives permission to access data in other app,steps to grant perm is called delegation authorization.
* OAuth token is “access card” at office/hotel.
* OAuth has below grant-type flows

Authorization code – only wen end-user is involved

PKCE

Client cred

Device code

Refresh token

Implicit flow(legacy)

Password grant(legacy)

Access/ID/refresh tokens

* Jargaons

Resource owner- end user

Client – canvas(ph edtor website)

Auth server – google auth server

Resource server – google photos server(some has same reso/auth server) same.

Scopes – scopes like read/wrte etc..

Eg: Slack has to get the client-Id and client secret from google API,later they will be used by google auth-server as which client is trying to connect

* Google auth server issues access,refresh tokens to PhotoEditor website and PE saves them for future interaction with google servers
* Client request to auth-server will have below details

Client\_id,redirect\_url,scope,state and response\_type

* Client,after getting auth-code from auth-server, will make a request to auth-server for token with below values

Code,

Client id & secret

Grant-type

Redirect\_uri

* oAuth2 playground
* PKCE – proof key for code exchange

When pub clients request access-tokens, some security concerns are raised,as public clients cannot securely store client-secret,due to this oauth provides a version of auth-code flow for pub client apps which makes use of PKCE.

* A)Client app on click, generates cryptographic random code\_verifier and generates a code\_challenge

b)code\_challenge is a base64,URL encoded string of SHA256

c)redirects user to auth server along with code\_challenge

d)auth-server stores code\_challenge and redirects user back to app with auth-code,for one use

e)client app sends auth code and code verifier to auth server

f)auth server verifies code\_challenge and code\_verifier,if they are valid respond with ID token and access-token

* Client Credentials grant type

When client make req to auth-server,it sends client-id,secret ,scope and grant-type(with value client-credentials) which indicates that we want to follow client-credentials grant type

* Refresh token grant type

Client reaches to resource server,with access token

RS responds with 403,as access token is expired

Client gives auth server refresh token and asks for new access token

Auth server checks for refresh token and gives new access token

Client reaches to resource server,with new access token

Resource server, responds with success

Req to auth-server contains the below –

Clientid,client\_secret, refresh\_token,scope,grant\_type

* Resource server token validation

1) validating access tokens remotely – tokens will be opaque token

2) validating access tokens locally – here no interaction between RS and AS, as RS relies on JSON web key set(JWKS),in JWKS auth server using a private key generates access token and using public key,RS will validate token.

* **Spring-Data** is parent project that contains sub-projects like jpa,jdbc,ldap etc

Spring-data-JPA internally uses hibernate framework

* Whatever persistence tech we use,SD provides common set of interfaces, you extend to define pers capab

Central interface in SD repository abstraction is repository,it’s a marker interface.

Public interface Repository<T,ID>

T – table(domain type) i.e., entity of table should be provided

ID – primary key inside table/entity

* CRUD Repository for interacting with db,

Public interface CrudRepository<T,ID> extends Repository<T,ID>

This provides developer with all methods needed to do crud-operations,

findById – using primary key

findAllById – any values

ListCRUDRepository – provides set of methods that returns entityClasses in a list,

* PagingAndSortingRepository – provides methods to retrieve entities using paging and sorting abstraction.

ListPagingAndSortingRepository,extension to PagingAndSortingRepository returning list instead of iterable wherever applicable.

* Spring-data-jpa is one of subproject of spring-data and internally leverages hibernate(ORM)

Interface JpaRepository extends ListCrudRepository<T,ID>, ListPagingAndSortingRepository<T,ID>

<dependency>

<artifactid> spring-boot-starter-data-jpa</artifactdid>

</dependency>

* @Entity

@Table(name = “contact\_msg”)

Public class Contact extends BaseEntity{

@Id

@GeneratedValue(strategy = GenerationType.AUTO,generator = “native”)

@GenericGenerator(name = “native”,strategy = “native”)

@Column(name = “contact\_id”)

Private int contactId;

@Repository

Public interface ContactRepository extends CrudRepository<Contact,Integer> - during runtime spring will create bean for interface and also its implementation.

@SpringBootApplication

@EnableJpaRepositories(“com.eazyschool.repository”)

@EntityScan(“com. eazyschool.model”)

Public interface ContactRepository extends CrudRepository<Contact,Integer>

Eg: class ContatcService{

@Autowired

Private ContactRepositiry conatctRepository;

Boolean saveDetails(Contact contact){

Boolean isSaved = true;

Contact saveContact = conatctRepository.save(contact);

Return isSaved;

}

* @MappedSuperClass – all entities extending this class should consider the fielfs also as part of their entity

@MappedSuperClass

Classs BaseEntity{

Private LocalDateTime ldt;

Private String date;

Private LocalDateTime ldt2;

Private String updatedBy;

@Enumerated(EnumType.STRING) – here it inserts string type to database

Private Type type;

Public enum Type{ FESTIVAL,FEDERAL}

**Derived-Query-Methods**

With spring-data-jpa, we can write method names to derive query and fetch data without writing code manually,Spring-data jpa will create queries and implementation at runtime by parsing these method names,

Eg: List<Person> findByLastName(String lastName);

Person findbyEmail(String email);

Public interface ContactRepository implemenst CrudRepository<Contact,String>{}

Person findByEmailAndLastName(String email,String lastName);

Eg: List<Contacts> list = contactRepository.findByStatus(Eazycontatcs.OPEN);

**Properties:**

Spring.jpa.show-sql = true; - to print sql statemsts in console

Spring.jpa.properties.hibernate.format\_sql = true;

Create conn,close conn,handle exception,transactions,provide sql statement,running sql statement,iterating results, transfer saving database data into pojo class – all are done by JPA.

* Introducer clause like find,read,query,count or get connected by/distinct with criteria clause like readBy,getBy,countBy ,findDistinct etc..

**Auditing**

@CreatedDate,@CreatedBy,@LastModifiedDate,@LastModifiedBy

@EnableJPAAudiitng(audiotrAwareRef=”auditAwareImpl”)

@EntityListeners(AuditingEntitiyListener.class)

**Validations**

@Documented – if used then documentation will be carry-forwarded by the annotation

**One-One Relation**

@Data

@Entity

Public class Person{

@OnetoOne(fetch = FetchType.EAGER,cascade=cascadeType.ALL,targetEntity=Address.class)

@JoinColumn(name = “address\_id”,referenceColumnName=”addressId”,nullable=true)

Private Address address;

* Fetch type can be EAGER or LAZY, i.e. for LAZY whenever we call using person,getAddress(),on the contrary EAGER will load its relation entities as well.
* By default all ToMany are lazy and ToOne are eager
* JPA allows entity state change from parent to child, this is called cascading in JPA.

CascadeType.PERSISIT

CascadeType.MERGE

CascadeType.REFRESH

CascadeType.REMOVE

CascadeType.DETACH

CascadeType.ALL

CascadeType from parent to child is valid, the otherway is invalid, also der is no default type

* If @OneToOne is mentioned in Person.java, then its uni-directional orelse if its mentioned in both Tables, then its bi-directional
* If nullable is false, then its optional relation or else it’s a mandatory relation between both of them

Authentication

Write custm logic to authemticate user instead of using in-memory authentication

@component

Class EazySchoolProvider implements AuthenticationProiver {

@override

Public Authentication authenticate(Authentication authentication){

}

* Encoding is converting data from 1 form to ther without any crytigraphy,it is reversible and it cantbe used for securing data,Eg: ASCII,base64 etc
* Encryption is process of transforming data,to guarantee confidentiality,it needs a secret, which we call as “key”, it is reversible by decryption with help of key,as long as key is confidential,encryption can be considered as secure
* Hashing , data is converted to hash value using some hash function,non-reversible,compare user entered hash value with hashvalue stored in db and verify
* Spring-security uses passwordEncoders,for hashing like

NoOpPasswordEncoder

StandardPasswordEncoder

Pbkdf2PasswordEncoder

BCryptPasswordEncoder

SCryptPasswordEncoder

* Spring.jpa.properties.javax.persistence.validation.mode = none, this will disable the javax bean validations by Spring-Data JPA

**Many to One-**

@Entity

Public class Person extends BaseEntity{

@ManyToOne(fetch = FetchType.LAZY, optional = true)

@JoinColumn(name=”class\_id”,referenceColum = “classId”,nullable = “true”)

Private EazyClass eazyClass;

Child-entity that has join column is called owner of relation,@JoinClolumn is used to specify foreign key column details

**One to Many -**

@Entity

Public class Person extends BaseEntity{

@OneToMany (mappedBy = “eazyClass”,fetch = FetchType.LAZY, cascade =cascade.PERSIST, targetEntiry=Person.class,optional = true)

Private Set<Person> person;

Private EazyClass eazyClass;

It also declares,mappedBy element to indicate the entiry that owns bidiretctonal relation,usually child entity is the one that owns relation and parent contain @OneToMany annotation.