**Ինչ է Spring MVC – ն (v6․0․4 02․02․2023 դրությամբ)**

Spring Framework – ի կոմպոնենտ է որը թույլ է տալիս ստեղծել web app – ներ Java – ով։

App – ները ստեղծվում են Modev-View-Controller կոնցեպտի հիման վրա։

MVC – ն պատտերն է որը ունի 3 կոմպոնենտ՝

Model – տվյալների հետ աշխատելու լոգիկա

View – user -ի համար տեսանելի հատվածի լոգիկա (օր HTML), ինտերֆեյս

Controller – նավիգացիայի լոգիկա, request – ների մշակում

Spring MVC- ում կիրառելի են Spring Core – ի բոլոր հնարավորությունները։

Spring MVC – ն կազմված է Java կլասսներից և դրանցում ակտիվ օգտագործվող անոտացիաներից, HTML – ներից JS կոդի կիրառմամբ, կոնֆիգուրացիոն ֆայլերից Java/Anotation/Xml

Http request – ի և Controller – ների միջև կան նաև DispatcherServlet որը հանդիսանում է Spring MVC app – ի մուտքի կետը և որը Spring տրամադրում է ավտոմատ։ Request – ը սերվերին փոխանցվելուց հետո փոխանցվում է DispatcherServleti – ի որը այն ուղարկում է ճիշտ controller – ին։ Նշված մոդելը կոչվում է առաջնային controller – ի շաբլոն որտեղ կենտրոնական սերվլետը DispatcherServlet – ը տրամադրում է ընդհանուր ալգորիթմ request – ների մշակման համար իսկ ավելի նեղ մշակումը կատարում են առանձին կոմպոնոնտները (controller - ները)։ Սերվլետների նման DispatcherServlet – ի համար նույնպես պետք է mapping` Java – ի կամ web.xml – ի միջոցով:

AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();

context.register(AppConfig.class);

// Create and register the DispatcherServlet

DispatcherServlet servlet = new DispatcherServlet(context);

ServletRegistration.Dynamic registration = servletContext.addServlet("app", servlet);

registration.setLoadOnStartup(1);

registration.addMapping("/app/\*");

web.xml – ի դեպքում՝

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/app-context.xml</param-value>

</context-param>

<servlet>

<servlet-name>app</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value></param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>app</servlet-name>

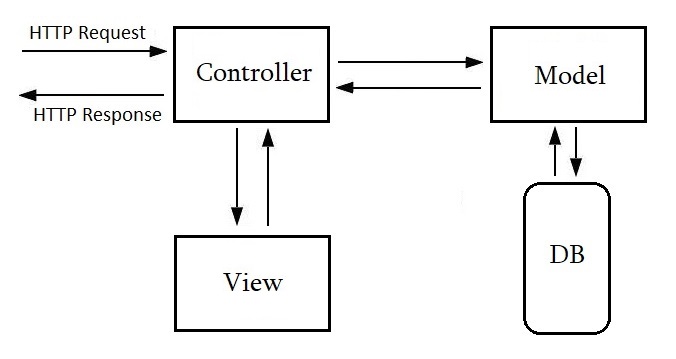
<url-pattern>/app/\*</url-pattern>

</servlet-mapping>

Controller- ը մշակում է request – ը, տվյալներով փոխանակվում model – ի հետ, user – ին ցույց է տալիս ճիշտ view, ըստ անհրաժեշտության վերահացեագրում է դեպի այլ ռեսուրսներ։

Model – ը պահում է իր մեջ տվյալներ, աշխատում է DB – ի հետ, controller – ին է տրամադրում տվյալներ։

View – ն տվյալները ստանում է controller – ից և արտապատկերում է դրանք բրաուզերում, HTML – ի դինամիկ գեներացման համար կարող են օգտագործվել շաբլոնիզաորներ օր՝ Thymeleaf, Freemarker, Velocity:

****

**@Controller** – ժառանգում է @Component անոտացիայից, այսինքն @ComponentScan – ը Controller – ի հետ աշխատում է նույն մեթոդով, եթե գտնում է անոտացիան ստեղծում է դրա bean – ը։ @Controller կլասսը կարող է իր մեջ պարունակել մեթոդներ ցանկացած անունով, որոնք համապատասխանում են 1 url – ի և սովորաբար վերադարձնում են String (պարտադիր չէ)։ Թե որ մեթոդը պետք է մշակի կոնկրետ request – ը որոշվում է mapping – ի միջոցով, որը սահմանվում մեթոդի վրա է HTTP request մեթոդներին համապատասխան անոտացիաներով՝ **@GetMapping, @PostMapping, @PutMapping, @DeleteMapping, @PatchMapping:** Երբեմն կարելի է հանդիպել հին սիտաքսին՝ **@RequestMapping(value = “/url” method = RequestMethod.GET):**

@RequestMapping – ը կարող է լինել նաև կլասսի վրա՝

@Controller

@RequestMapping(“/someUrl”)

public class MyController {}

Այս դեպքում մնացած բոլոր mapping մեթոդներըի հասցեները պետք է իրենց մեջ պարունակեն RequestMapping – ում նշված արժեքը։

Mapping – ը կապում է controller – ը հասցեի հետ որին կարելի է դիմել։

Mapping մեթոդները String վերադարձնելիս default – ով փորձում են գտնել համապատասխան view – ն։ Օր Thymeleaf շաբլոնիզատորից օգտվելու համար պետք է հայտարարել համապատասխան bean – ները և web.xml – ում պետք է նշվի նաև DispatcherServlet – ի mapping – ը և այն xml – ը որից պետք է վերցվեն մնացած config – ները (bean - երը)։ Նույն գործողությունը առանց xml և web.xml օգտագործելու կարելի կատարել միայն Java կոդով։ Այս դեպքում պետք է ժառանգել AbstractAnnotationConfigDispatcherServletInitializer աբստրակտ կլասսից որի մեթոդների միջ համապատասխան փոփոխություններից հետո այն կփոխարինի web.xml – ին։

@Controller անոտացիայով կլասսներում Իսկ ուղղակի String ստանալու համար Mapping մեթոդին պետք է ավելացվի @RequestBody անոտացիան։

Դա փոխարեն կարելի է օգտագործել @RestController անոտացիան որը իրենից ներկայացնում է @Controller + @ResponseBody բոլոր մեթոդների վրա:

Get – ի միջոցով փոխանցված պարրամետրերի մշակումը Controller – ում կարող է կատարվել HttpServetRequest օբյեկտի getParameter(“name”) մեթոդի կամ @RequestParam(“name”) անոտացիայի միջոցով`

@RequestParam(“name”) String name

Այս դեպքում Spring – ը ավտոմատ կվերցնի պարամետրը url – ից և կվերագրի փոփոխականին։ HttpServetRequest օբյեկտ օգտագործելիս եթե նշված անունով պարամետրը չի փոխանցվում ապա կունենանք null իսկ @RequestParam անոտացիայի դեպքում` 400 error: Եթե չենք ուզում նման վարքագիծ ապա կարելի է՝ @RequestParam(value=”name”, required = false):

View – ին (Html - ին) կարելի է տվյալներ փոխանցել նաև Model օբյեկտի addAttribute(key, value) միջոցով: շաբլոնիզատորները հնարավորություն են տալիս Html – ում expression language – ի միջոցով ստանալ ատրիբուտները։

**CRUD, REST, DAO**

**CRUD (**create, read, update, delete) սրանք տվյալների հետ աշխատելու բազային ֆունկցիաներն են։ Այսինքն CRUD application – ը հնարավորություն է տալիս ստեղծել, կարդալ, փոխել, ջնջել տվյալներ DB – ում։ Crud application – ները որպես կանոն նաև web application – ներ են, այսինքն նշված գործողությունները իրականացվում են HTTP – ի միջոցով՝ create – POST, read – GET, update – PATCH, delete - DELETE։

CRUD app – ների համար կան convension – ներ՝

GET /entities – բոլորը ստանալու համար (read)

POST /entities – նորը ստեղծելու համար (create)

GET /entities/new – նորը ստեղծելու HTML ֆորման ստանալու համար

GET /entities/:id/edit – կոնկրետ մեկի փոփոխման ֆորման ստանալուհամար

GET /entities/:id – կոնկրետ մեկը ստանալու համար

PATCH /entities/:id – կոնկրետ մեկը փոփոխելու համար

DELETE /entities/:id – կոնկրետ մեկը ջնջելու համար

Այս ամենը նկարագրում է REST – ը, որը իրենից ներկայացնում է web ծրագրավորման պատտերն։ REST – ի միջոցով նկարագրվում է թե ինչպես պետք է փոխազդեն կլիենտը և սերվերը՝ HTTP – ի միջոցով։ Փոխազդեցությունը որպես կանոն տեղի է ունենում վերոհիշյալ 4 գործողությունների միջոցով։

CRUD – ը հիմնական մեթոդներն են որոնք օգտագործվում են տվյալների հետ աշխատելիս իսկ REST – ը պատտերն է որը նկարագրում է թե ինչ HTTP մեթոդներով են փոխազդելու սերվերը և կլիենտը այդ գործողությունները կատարելու համար։

Այսինքն CRUD – ը պարտադիր չէ որ լինի web app այն կարող է լինելե պարզապես app որը աշխատում է տվյալների հետ, իսկ REST – ի դեպքում արդեն խոսքը վերաբերում է HTTP պրոտոկոլով աշխատանքին՝ այսինքն web app – ին։

Նպատակահարման չէ DB – ի հետ աշխատանքի լոգիկան գրել ամբողղջությամբ model – հատվածում, այդ պատճառով օգտագործվում է DAO (data acsess object) պատտերնը։ DAO պատտերնը ենթադրում է կոնկրետ entity – ի համար DB – ի հետ աշխատանքի լոգիկան դուրս բերել առանձին կլասսի մեջ։ DAO – կլասսները սովորաբար պարունակում են sql կոդ։

@ModelAttribute անոտացիան կարող է օգտագործվել մեթոդների կամ արգումենտների հետ շաբլոնիզատորների հետ աշխատանքի ժամանակ։

Մեթոդի հետ այս անոտացիան նշանակում է որ model տեսակի օբյեկտի մեջ պետք է ներդրվի անոտացիայի արգումենտում նշված key – ով այն արժեքը որը վերադարձնում է անոտավորված մեթոդը։

Մեթոդի արգումենտի հետ կարող է կիրառվել օրինակ ֆորմաներից ստացով տվյալների հիման վրա օբյեկտ ստեղծելիս։

Վալիդացիաների համար կարող է կիրառվել hibernet validator dependency – ն։ Որը string – երի համար ունի օր` @NotEmpty(message = “error message”), @Size(min = 2, max = 30, message = “error message”), @Email(message = “error”), @Pattern(regexp = “”, message = “”)։ Թվերի համար օր՝ @Min(value = 0, message = “error message”):

Այլ վալիդացիաներ՝ <https://alexkosarev.name/2018/07/30/bean-validation-api/>

Սրանից հետո @Valid անոտացիան դրվում է համապատասխան օբյեկտի հետ արգումենտում, իսկ error – ը գրվում է BindingResult օբյեկտի մեջ (spring validation – ի նման)։

**JdbcTemplate**

JdbcTemplate – ը wrap է անում JDBC Api – ը և հեշտացնում է աշխատանքը DB – ի հետ։ Jdbc Api – ի խնդիրներից կարելի է նշել մեծ ծավալի կոդը (connection, statement, resultset), կոդի կրկնությունը (օր preparedStatement – ում անընդհատ set անելը), SQLException – ը որը քիչ ինֆորմատիվ է throw է արվում ցանկացած խնդրի դեպքում և պետք է այն միշտ catch անել։

Template – ով աշխատելու համար պետք է spring-jdbc dependency – ն։ Օգտագործելու համար պետք է ստեղծել դրա bean – ը, որի համար անհրաժեշտ է նաև ստեղծել DataSource bean: Վեջինի միջոցով նշվում են DB – ի config -ները՝ setDriverClassName, setUrl, setUsername, setPassword։ Դրանից հետո արդեն ստեղծել JdbcTemplate bean – ը DataSource – ը փոխանցելով որպես արգումենտ։ Բայց կոդի մեջ բաց նշել DB տվյալները վատ պրակտիկա է քանի որ կոդին հասանելիություն ունեցողը կարող է միանալ DB – ի։ Սրանից խուսափելու համար DB config – ները նպատակահարմար է տալ արտաքին ֆայլից։ Config – ները կարելի է key = value տեսքով գրել resources – ում database.property ֆայլում որը կարելի է ignore անել push – ի ժամանակ, իսկ արժեքները կոդում bean ստեղծելիս նշել key - երով։ Սրա հետ մեկտեղ կարող է լինել նաև database.properties.origin ֆայլ որը իր մեջ պահում է միայն key= – երը առանց արժեքների։ Այս ֆայլը public է և նախատեսված է նրա համար որ պրոյեկտից օգտվողը իմանա թե ինչ տվյալներ են պահանջվում DB – ի հետ աշխատելու համար, որպես հուշում։

JdbcTemplate – ի (կախված sqlQuery – ի տեսակից) query կամ update մեթոդին փոխանցում է sql query – ն և որպես 2-րդ արգումենտ այսպես կօչված row mapper որը տվյալների բազայում աղյուսակի պարունակությունը վերագրում է օբյեկտին։ Row mapper – ը կլասս է որը իմպլեմենտացնում է RowMapper ինտերֆեյսը և override անում mapRow(ResultSet rs, int i) մեթոդը (generic է, նշվում է աղյուսակին համապատասխան մոդելի օբյեկտը, Comparator – ի նման): Նշված մեթոդից պետք է վերցնել resultSet օբյեկտը և այն վերագրել մոդել կլասսին։

JdbcTemplate – ը ավտոմատ օգտագործում է prepared statement: Պարրամետրերը փոխանցվում են sqlQuery – ին հաջորդող արգումենտով որը օբյեկտների մասիվ է օր՝

query(“SELECT \* FROM Person WHERE id=?”, new Object [] {id}, new MyRowMapper);

query – ն վերադարձնում է list:

update – ի դեպքում ի տարբերություն query - ի պարրամետրերի արգումենտը Object … args է։ Պատճառը այն է որ query – ի ժամանակ փոխանցվում է նաև 3 – րդ արգումենտ։

Եթե մոդել օբյեկտի դաշտերի անունները համընկնում են աղյուսակի սյուների անունների հետ ապա custom row mapper գրելու կարիք չկա, փոխարենը կարելի է օգտագործել spring – ի տրամադրած BeanPropertyRowMapper<>() օբյեկտը որին որպես արգումենտ փոխանցվում է մոդել կլասսը MyModelClass.class։

butchUpdate – ի համար JdbcTemplate – ը ունի նույնանուն մեթոդ, որը որպես արգումենտ ստանում է sql query – ն և 2 – րդ արգումենտ BatchPreparedStatementSetter օբյեկտ որը կարելի է ռեալիզացնել անանուն կլասսով և override անել դրա setValues(preparedStatment, int i), getBatchSize։ Վերջինի մեջ վերադարձվում է batch – ի size – ը, իսկ setValues – ը որպես արգումենտ ստանալով preparedStatment՝ դրա set մեթոդներով կարող է կատարլ արժեքների վերագրումները։ setValues - ի 2-րդ i արգումենտը ինդեքսն է օր՝

preparedStatement.setInt(1, peopleList.get(i).getId());

update – ի ու batchUpdate – ի տարբերությունը շատ մեծ է օր 1000 տողի համար 14450 մվրկ և 46 մվրկ։

**Spring Boot**

Spring Boot – ը Spring Framework – ի կոմպոնենտ է։

Spring framework – ը ունի մի շարք անհարմարություններ՝ config – ների համար մեծ քանակույամբ կոդ, dependecy – ները և դրանց համատեղելիությունը շատ հաճախ դրանց մեջ լինում են կոնֆլիկտներ, app – ի deployment – ը (այդ թվում tomcat – ի ձեռքով configuration - ը)։ Spring boot – ը լուծում է նշված և այլ խնդիրներ։ Կարող է ավտամատ կատարել coniguration – ը օգտագործվող տարբեր գործիքների համար օր՝ (Hibernate, Security, Timeleaf և այլն)։ App – ի war ֆայլում հավաքվելը և deploy – ը հեշտանում են այնքանով որ Spring boot – ը իր մեջ պարունակում է web սերվեր ընդ որում կարելի է ընտրել սերվերը tomcat կամ այլ։ Այսինքն app – ն run է լինում main մեթոդից սովորական java application – ի նման, իսկ մնացած գործողությունները տեղի են ունենում տակից։ Dependency – ներ դեպքում Spring Boot – ը տրամադրում է այսպես կոչված starter – ներ, որոնք իրենցից ներկայացնում են dependency – ներ խմբավորված մեկ մեկ dependency – ի մեջ։ Նման մոտեցումը թույլ է տալիս խուսափել շատ dependency – ների միջև անհամատեղելիություններից։ Օր՝ spring-boot-starter-web որը իր մեջ պարունակում է sping core, web, web-mvc, json – ի հետ աշխատող կախվածություններ, tomcat և այլն։

Բացասական կողմը այն է որ տակից տեղի ունեցող գործողություններ են շատ և որպես հետևանք error – ների ծագումը/պատճառների բացահայտումը կարող է բարդանալ։

**Rest API**

Spring app – ն լինելով հիմնականում backend app ընդունում է request – ներ և վերադարձնում է տվյալներ հիմնականում json – ի տեսքով։

Այն api – ները որոնք ընդունում են request – ներ և վերադարձնում response – ներ կոչվում են Rest Api որին հոմանիշ են RestfulApri, սերվիս, backend, ուղղակի API: Այսինքն view – ի հատվածը frontend հատվածում է։

Rest Api – ները ավելի տարածված են հատկապես միկրոսերվիսային համակարգերի պատճառով, որոնք իրար հետ շփվում են Rest – ի միջոցով։

Rest Api – ի կոնտեքստում կարևոր է նաև Jackson գրադարանը: Java – ում ի տարբերություն JS – ի JSON ֆորմատը հնարավոր չէ միանգամից օգտագործել, կարիք կա այն վերածել Java օբյեկտի: Json – ից java օբյեկտ և հակառակը կոնվերտացիաների համար օգտագործվում է Jackson գրադարանը: Jackson – ը աշ խատում է java bean ստանդարտի օբյեկտների հետ այսինքն հիմնվում է get, set մեթոդների վրա: Spring framework – ը response – ի ժամանակ default օգտագործում է jackson – ը, այսինքն եթե օրինակ mapping մեթոդը վերադարձնում է որևէ օբյեկտ այն ավտոմատ կկոնվերտացվի JSON – ի և այդ տեսխով կուղարկվի: Jackson օգտագործելու համար պետք է դրա dependency – ն:

**RestTemplate**

Արտաքին ռեսուրսներին call անելու համար օգտագործվում է RestTemplate – ը որ ստանում ենք spring-web dependency – ից:

Օբյեկտի T getForObject(String url, Class<T> responseType) մեթոդեշով կարելի է ստանալ պարամետրում նշված տիպի օբյեկտը: 3-րդ Պարամետրում կարելի է փոխանցել նաև request պարամետրեր HashMap – ի միջոցով:

postForObject(String url, Object requestBody, Class<T> responseType) – ի դեպքում նշվում է նաև requestBody – ն (կոնվերտացվում է json - ի), որը հարմար է նշել HashMap – ի տեսքով քանի որ այն շատ նման է Json – ի key – value տեսքին, բայց HashMap – ը ցանցով ուղարկելու համար այն պետք է ներդնել HttpEntity օբյեկտի մեջ: Օր՝ new HttpEntity<>(hashMap);

header – ներ փոխանցելու համար կա HttpHeaders օբյեկտը և դրա set կամ add մեթոդներով ավելացնել header – ներ

**@RequestBody**

Օգտագործվում է post – ի ժամանակ։ Կլիենտի ուղարկած json – ը ավտոմատ վերածվում է օբյեկտի jackson – ի միջոցով և վերագրվում անոտացիայով նշված օբյեկտին։

**Hibernate**

Hibernate – ը ինքնուրույն գրադարան է և կախված չէ Spring Framework – ից:

Կառուցվածքը կարելի է բաժանել հետևյալ մակարդակների՝

* Java app
* Hibernate framework layer (core objects)
* Backhand layer – այն api – ն որը աշխատում է hibernate – ի տակից (JDBC, JNDI, JTA)
* Database layer

Քանի որ java կլասսները և աղյուսակները DB – ում իրար շատ նման են ստեղծվել է ORM (Object-relational Mapping) գաղափարը: Hibernate – ը իրենից ներկայացնում է ORM – ի ռեալիզացիա որը իր վրա է վերցնում աղյուսակների ավտոմատ convert – ը java օբյեկտների և հակառակը, բայց Hibernate – ը ընդամենը կարգավորում է իսկ տակից միևնույն է աշխատում է JDBC API – ն:

Hibernate օգտագործելիս կարիք չկա գրելու sql կոդ: Hibernate- ը ունի գործողութոյւններին համանուն մեթոդներ օր՝ save որոնք տակից կատարում են sql query – ի գործողությունները:

Աղյուսակների միջև կապերը java կլասսի մեջ կունենան հետևյալ տեսքը՝ Parent\_Table – ի տողը նկարագրող կլասսը իր մեջ կպահի child – երի տողերի լիստ, իսկ child – ի տողը իր մեջ parent – ի տողի տիպի օբյեկտ: Այս դեպքում նշված դաշտերը get անելիս Hibernate – ը տակից ավտոմատ կաշխատեցնի join query – ն և կվերադարձնի իրար հետ կապված տվյալները:

Hibernate – ի առացքային օբյեկտներն են՝

SessionFactory, Session, Transaction, ConnectionProvider, TransactionFactory

Hibernat – ում օգտագործվող օբեյկտներից են՝

Session session = sessionFactory.getCurrentSession();

Օբյեկտը ունի save, update, get և այլն մեթոդներ որոնց միջոցով ապահովում է անհրաժեշ աշխատանքային ինտերֆեյսը app – ի և DB – ի միջև: Հանդիսանում է JDBC – ի wrapper և իր մեջ պահում է նաև քեշ: Ունի նաև մեթոդներ տրանզակցիաների համար:

Մինչև այդ մեթոդների կանչը պետք է session օբյեկտի վրա կանչել beginTransaction() – ը, իսկ ավարտելուց հետո՝ getTransaction().commit(); Տրանզակցիաները ապահովում են միասնական աշխատանքը: Transaction օբյեկտը ունի համապատասխան մեթոդներ տրանզակցիաների կառավարման համար:

Commit և rollback – ը աշխատում են նույն սկզբունքով, ինչ որ JDBC - ում

Տրանզակցիաները կարող են նաև արգելափակել DB – ի հասանելիությունը, օր՝ քանի դեռ insert է արվում select մի արա, կամ քանի դեռ update է արվում ուրիշ update թույլ մի տուր և այլն:

Արգելափակման պրոցեսը ունի տարբեր մակարդակներ (Transaction Isolation Level) և շատ նման է thread – ների դեպքում synchronize – ի ֆունկցիոնալությանը:

Hibernate – ով աշխատելու համար պետք է hibernate core գրադարանը:

Spring – ից անկախ աշխատելու դեպքում կարիք կա տալ Hibernate – ի config – ներ resources – ում hibernate.properties – ի մեջ: Ֆայլի մեջ նշվում է`

hibernate.driver.class=com.microsoft.sqlserver.jdbc.SQLServerDriver  
hibernate.connection.url=jdbc:sqlserver://localhost:1433;database=TestHibernate  
hibernate.connection.username=  
hibernate.connection.password=  
  
hibernate.dialect=org.hibernate.dialect.SQLServerDialect  
hibernate.show\_sql=true  
hibernate.current\_session\_context\_class=thread

Դրանից հետո պետք է ստեղծել աղյուսակին համապատասխան java կլասս և դրա վրա դնել **@Entity** անոտացիան, որը ցույց է տալիս որ կլասսը կապված է DB – ի հետ: Կարելի է ավելացնել նաև @Table(name=tablename), բայց սրա կարիքը չկա եթե կլասի և աղյուսակի անունները համընկնում են:

@Entity – ով կլասսները պարտադիր պետք է ունենան դատարկ կոնստրուկտոր և id դաշտ որը պետք է նշվ @Id անոտացիայով:

Դաշտերը պետք է անոտավորվեն @Column(name = columnName)

Hibernate – ի Configuration տիպի օբյեկտում նշվում է model class – ը, որով Hibernate – ը հասկանում է որ DB- ում կա Person աղյուսակ որի հետ պետք է աշխատել: Այս ընթացքում ավտոմատ միանում է նաև hibernate.properties - ը

Configuration configuration = new Configuration().addAnnotatedClass(Person.class);

Configuration – ից կարելի է ստանալ SessionFactory

SessionFactory sessionFactory = configuration.buildSessionFactory();

Session session = sessionFactory.getCurrentSession();

session.get(Class clazz, int id);

session.save(Object o);

Որպեսզի կլասսի մեջ id – ի արժեքը ստացվի DB – ի գեներացրաց արժեքին համապատասխան պետք է @GeneratedValue(strategy = GenerationType.IDENTITY) անոտացիան, սրանով Hibernate – ին ասվում է որ պետք չէ ձեռք տալ այս դաշտին դրա արժեքը գեներացվում է DB – ում: save – ից անմիջապես հետո կարելի է ստանալ դրա id քանի որ Hibernate – ը ավտոմատ այն կավելացնի օբյեկտի մեջ save – ի հետո:

GenerationType.SEQUENCE -կարող է օգտագործվել երբ DB – ն չունի id գեներացնելու ֆունկցիա և id – ն պետք է գեներացվի java – ում:

Update – ի համար պետք է get – ով ստանալ տողը java – ում դրա օբյեկտի վրա կատարել անհրաժեշտ փոփոխությունները set մեթոդների միջոցով և վերջ տրանզակցիայի commit – ից հետո փոփոխությունները կանցնեն DB:

Delete – ի համար նորից get – ի միջոցով ստանալ օբյեկտը և փոխանցել session.remove(Object o) մեթոդին:

**HQL**

Hibernate – ում query գրելու համար օգտագործվում է HQL սինտաքսը:

Query գրելու կարիք կարող է առաջանալ սպեցիֆիկ գործողություններ կատարելու համար, այն դեպքորում երբ անհրաժեշտ գործողության համարժեք մեթոդը Hibernate – ը չունի, օր՝ ստանալ տվյալները որևէ պայմանով (age > 20), կամ ջնջել տողերը որոնցում անունը սկսվում է որևէ տառով, update անել բոլորին և այլն:

SQL – աշխատում է աղյուսակների հետ DB – ում, իսկ HQL ոչինչ չգիտի աղյուսակների մասին և աշխատում է java entity – ների հետ: Մնացած հարցերում SQL – ը և HQL – ը շատ նման են: Օր՝

SQL – SELECT \* FROM Person WHERE name = ‘Test’

HQL - FROM Person WHERE name = ‘Test’

HQL – ը ստեղծվել է որպեսզի աղյուսակի մակարդակ իջնելու կարիք չլինի և աշխատենք java entity – ների հետ:

HQL- ը տակից կոնվերտացվում է SQL – ի

Hibernate- ում կարելի է HQL – ի փոխարեն օգտագործել նաև սովորական SQL

Գրելու համար` session.createQuery(String query).getResultList()

Update – ի համար ՝ ` session.createQuery(String updateQuery).executeQuery()

Օր՝

currentSession.createQuery("UPDATE Person SET name = 'Test' WHERE age > 30").executeUpdate();

List<Person> people = currentSession.createQuery("FROM Person WHERE name LIKE 'J%'").getResultList();

currentSession.createQuery("DELETE FROM Person WHERE name LIKE 'J%'").executeUpdate();

**Hibernate entity – ն ունի կյանքի հետևյալ ցիկլերը՝**

* **Transient –** երբ ստեղծվում է entity – ի օբյեկտը և դրա համար դեռևս չի կանչվել save մեթոդը այն գտնվում է այս state – ում, այսինքն այն դեռևս կապված չէ Hibernate – ի հետ և դրա կողմից track չի լինում, չի գտնվում է այսպես կոչված Persistance context – ում: Այս օբյեկտի վրա setter – ների կանչը ոչ մի մերպ դեռևս չի ազդում DB – ի վրա: Այսինքն transient օբյեկտը կարելի է ընկալել որպես սովորական java օբյեկտ:
* **Persistent (Managed) –** երբentity օբյեկտի համար կանչվում է save մեթոդը և այն տարնզակցիայի commit() մեթոդով Hibernate – ի միջոցով save է լինում DB - ում օբյեկտը անցնում է այս state – ին: Օբյեկտը այս վիճակին է անցնում նաև երբ այն get մեթոդով ստանում ենք DB – ից: Օբյեկտը track է լինում Hibernate – ի կողմից և գտնվում է persistence context – ում: Այս օբյեկտների վրա կանչված setter մեթոդները գեներացնում են query և ազդում են DB- ի վրա: persistence context – ում օբյեկտը գտնվում է այնքան ժամանակ քանի դեռ աշխատում է Hibernate session – ն:
* **Detached** – բնութագրվում է նրանով որ օբյեկտը դուրս է գալիս persistence contex – ից և նորից դառնում է սովորական java օբյեկտ: Նման է Transient state – ին: Այս state – ին անցնելու համար` session.detach(entityObject): ԵՎ հակառակը կարելի է օբյեկտը վերադարձնել persistence context session.merge(entityObject) մեթոդով:
* **Removed –** այսstate – ին օբյեկտը անցնում է երբ այն commit() – ից հետո ջնջվում է սեսսիայից՝ session.remove(entityObject)

**Cascading – ը Hibernate – ում**

Կարող ենք կարգավորել այնպես որ օր՝ parent օբյեկտ ստեղծելուց և այն նոր child – օբյեկտի կոնստրուկտորի մեջ նշելոց հետո բավարար լինի save անել միայն parent – ը իսկ դրա հետ կապված child – ը արդեն save լինի cascading – ի միջոցով այլ ոչ թե առանձի դրա համար save մեթոդով: Կասկադավորումը սովորաբար կատարվում է parent – ում: @OneToMany անոտացիան ունի cascade պարրամետր որտեղ նշվում են կասկադավորման ենթակա գործողությունները:

Օր՝ CascadeType.Persist – նշանակում է save – ի ժամանակ

@OneToMany(mappedBy = ‘owner’, cascade = CascadeType.Persist)

Տարբերությունը այն է որ cascade ատրիբուտը օգտագործելիս session.save(Object savingObject) – ի փոխարեն պետք է լինի session.persist(Object savingObject): Մեթոդները իրար շատ նման են, բայց ունեն հետևյալ տարբերությունները՝

|  |  |
| --- | --- |
| svae | persist |
| Պատկանում է միայն Hibernate գադարանին և չկա այլ JPA գրադարաններում | Ստանում ենք JPA գրադարանից |
| Վերադարձնում է save եղած entity – ի primary key - ը | Void է և ոչինչ չի վերադարձնում |
| Քանի որ վերադարձնում է արժեք ապա Երաշխավորված է որ primary key – ը արդեն նշանակված կլինի միանգամից մեթոդի կանչից հետո | Նման բան չի երաշխավորում, այսինքն արժեքը ինչ որ պահի կնշանակվի բայց ոչ անմիջապես կանչից հետո |

save – ը օգտագործելու դեպքում cascade ատրիբուտի փոխարեն պետք է ավելացվի @Cascade(CascadeType.Save\_Update), արգումենտը ընդունում է vararg և մնացած տիպերը կարելի է նշել ստորակետով: Այս դեպքում persist() – չի կասկադավորվի իսկ save – ը կկասկադավորվի:

CascadeType.REFRESH – ը օգտագործվում է քեշը թարմացնելու համար

**Join - ը Hibernate – ում:**

Join – երի համար կան հատուկ անոտացիաներ որոնք դրվում են դաշտերի վրա՝

**OneToMany**

@OneToMany(mappedBy = '') – իրենից ներկայացնում է child – երի List parent աղյուսակում, mappedBy – ում նշվում է childTable – ում parentTable – ի օբյեկտի անունը:

@ManyToOne, @JoinColumn(name = '', referencedColumnName = '') – իրենից parent տիպի օբյեկտ child աղյուսակում, name – ում նշվում է childTable – ում foreign key – ի սյան անունը, referencedColumnName – ում նշվում է parentTable – ում id – ի սյան անունը: @JoinColumn – ն ով կլասսը կոչվում է նաև owning side այսինքն այն կողմը որ իր մեջ ունի foreign key:

Նշված անոտացիաներից հետո եթե կանչենք համապատասխան դաշտերի getter – ները hibernate – ը տակի կկատարի անհրաժեշտ join – ը և կվերադարձնի List: Նշված տարբերակը աշխատում է միայն տրանզակցիայի շրջանակում:

Hibernate – ը ունի քեշավորման մեխանիզմ և get – ից հետո հաջորդ get – ի ժամանակ արժեքը վերադրձվում է քեշից: Այսինքն օրինակ childTable – որևէ նոր օբյեկտ ավելացնելուց հետո պետք է դրա foreign key արժեքով տողը get անել parent աղյուսակից և դրա child – երի list – ի մեջ ավելացնել նոր ստեղծված օբյեկտը որը ավելացվել է childTable – ում:

Քեշի թարմացման համար կարելի է օգտագործել session.refresh() մեթոդը:

Child – ից parent – ի հետ կապված տողը կամ բոլոր տողերը ջնջելու համար բավական է ստանալ parent – ի մեջ սահմանված child - երի List – ը և դրանում ջնջել:

Hibernate – ում DB query – ները գեներացվում են միայն entity – ի getter մեթոդները օգտագործելիս, մնացած դեպքերում օր՝ childList.clear() query չի գեներանում:

**Spring security**

Spring framework – կոմպոնենտ է նախատեսվա անվտանգության համար: Օգտագործվում է ավտորիզացիոն գործողությունների համար և մոնոլիտ և միկրոսերվիսային համակարգերում:

Authentication – loggin / passwor այսինքն հատագա գործողությունները շարունակելու համար մուտք սերվիս

Authorization – Authentication – ից հետո որոշակի դեր օր՝ user, admin և այլն: Այսինքն դերերի բաժանումը դա authorization – ն է թե ինչ կարող ենք անել սերվիսում: Առանձին դերեր ավտորիզացված են մեկ իսկ այլ դերեր մեկ այլ գործողությունների համար (permission):

Security – ն աշխատում է ֆիլտրների միջոցով որոնք աշխատում են HTTP request – ի ու Spring application – ի միջև: Ֆիլտրը օբյեկտ է որը մշակում է requst – ները միջև դրանք կհասնեն controller – ին: Ֆիլտրները աշխատում են security գրադարանը ավելացնելուց հետո, boot – ում առանց configuration – ի անհրաժեշտության:

Authentication – ի ռեալիզացիա:

Կատարվում է AuthenticationProvider ինտերֆեյսի և Authentication օբյեկտի միջոցով: Պետք է ստեղծել ինտերֆեյսը ռեալիզացնող կլասս և դրա Authentication authenticate(Authentication credentials)

մեթոդի մեջ ռեալիզացնել անհրաժեշտ լոգիկան, որը կարող է ենթադրել օրինակ DB – ից տվյալների հետ համեմատում, authenticate – ի իրականացում այլ սերվերի վրա և այլն:

app – ի մեջ կարող է լինել 1 – ից ավելի AuthenticationProvider:

Մեթոդի վերադարձրած Authentication տիպի օբյեկտը իր մեջ պարունակում է principal` այն օբյեկտը որը ստանում ենք հաջող auth – ից հետո, սովորաբար user – ի օբյեկտը: Auth – ից հետո ամեն հաջորդ request – ի ժամանակ պետք չէ auth լինելքանի որ principal – ը պահվում է սեսսիայի մեջ: Սեսսիայից principal օբյեկտի load լինելու համար աշխատում է առանձին ֆիլտր: Սեսսիայի հետ աշխատանքը իրականացվում է cookie – ների միջոցով: Cookie – ն key:value ձևով պահվող տվյալներ են կոնկրետ ռեսուրսի համար, որոնք պահվում են բրաուզերում: Request – ի ժամանակ դրանք ուղարկվում են սերվերին: Սերվերը ավելացնում է բրաուզերում նոր cookie – ներ կամ փոփոխում է դրանք: ունեն նաև expires – ի աժեք և Domain, Path՝ Same Origin Policy – ի կոնցեպտում:

Spring Security – ն app – ին միացնելուց հետո մինչև request – ը կհասնի controller – ին բացվում է login page որում կարելի է մուտք գործել security – ի կողմից ավտոմատ գեներացված user – ով, default login – ը՝ user, password – ը գեներացվում է log – երում: