**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>

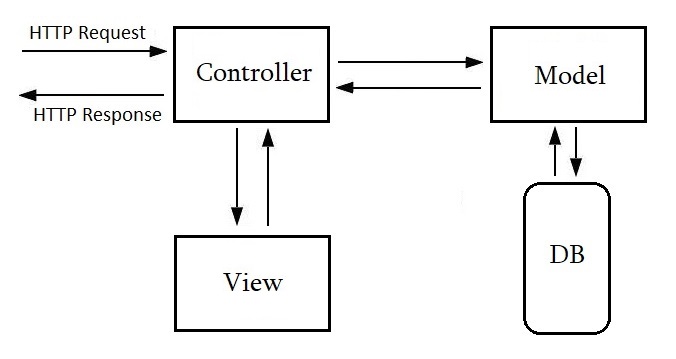
<url-pattern>/app/\*</url-pattern>

</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։ Model – օբյեկտները պետք է համապատասխանեն Java Bean ստանդարտին, քանի որ ավտոմատացված, տակից տեղի ունեցող գործողությունները (dat binding և այլն) աշխատում են getter setter մեթոդների հետ:

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:

****

**Անոտավորված controller - ներ**

**@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 մեթոդներում կարող են օգտագործվել path շաբլոններ որոնք ունեն նաև դրանց օբյեկտային նկարագրությունները **PathPatter** և **AntPathMatcher** տիպերում:

\* - ը նշանակում է ցանկացած բան, կարող է օգտագործվել հետևյալ կերպ՝

/resources/\*.png

/resources/\*\*

/resources/{project}/versions – {} – ով նշվում են path variable – ները որոնց mapping մեթոդի մեջ կարող ենք ստաալ **@PathVariable** անոտացիայով փոփոխականով, ընդ որում փոփոխականի և {} – ում նշված անունները պետք է լինեն նույնը կամ արգումենտում նշվի դրա անունը **@PathVariable('name')**: Թվերը ավտոմատ cast են լինում փոփոխականի տիպին: Կարող են օգտագործվել նաև regexp – ներով փոփխականներ path-ի մեջ:

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 կլասսների մեթոդներում վերադարձվող արժեքները ընդունվում են որպես view հատվածի հղում, այսինքն օրինակ html – ի անում: Իսկ ուղղակի String ստանալու համար Mapping մեթոդին պետք է ավելացվի @RequestBody անոտացիան որ HttpMessageConverter – ի միջոցով ResponseBody – ում ներառում է մեթոդի վերադարձրած արժեքը: @RequestBody – ն կարող է կիրառվել նաև կլասսի մակարդակում, այս դեպքում անոտացիան կվերաբերի կլասսի բոլոր մեթոդներին: Անոտացիայի փոխարեն mapping մեթոդի վերադարձրած տիպը կարող է լինել ResponseEntity<Object> որ իր մեջ պարունակում է body – ն և header – ները:

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

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։

**Mapping մեթոդների ատրիբուտները**

@PosrMapping անոտացիայում consumers = ''application/json'' – ով սահմանափակվում է @RequestBody – ի մեդիա տիպը: Տիպից առաջ կարող է նշվել նաև ժխտում !, այսինքն ամեն ինչ բացի դրանից:

@GetMapping անոտացիայում produces = ''''application/json '' – ով նշվում է մեդիա տիպը որը վերադարձնում է մեթոդը:

Mapping մեթոդների կանչը կարելի սահմանափակել եթե դրանցում բացակայում են request պարամետրերը, դրա համար օգտագործվում է params ատրիբուտը որը ստուգում է պարամետրի արժեքի համապատասխանությունը որոշակի արժեքի կամ դրա բացակայությունը կամ առկայությունը:

params=''myParam=myValue'', params=''myParam'', params=''!myParam''

Նույն տարամաբանությամբ աշխատում է headers ատրիբուտը header – ների համար:

Mapping մեթոդ կարելի է ավելացնել նաև @Configuration կլասսում՝

@Configuration

public class MyConfig {

@Autowired

public void setHandlerMapping(RequestMappingHandlerMapping mapping, UserHandler handler) throws NoSuchMethodException {

RequestMappingInfo info = RequestMappingInfo

.paths("/user/{id}").methods(RequestMethod.GET).build();

Method method = UserHandler.class.getMethod("getUser", Long.class);

mapping.registerMapping(info, handler, method);

}

}

**Mapping մեթոդների արգումենտների հետ օգտագործվող այլ անոտացիաներ**

**@MatrixVariable -** /somePath;q=11;r=22 առանձնացվում են ; - ով, իսկ փոփոխականի անունը վերցնում է համապատասխան անունով արժեքը: Մասիվով ստանալու համար օգտագործվում է՝

@MatrixVariable MultiValueMap<String, String> matrixVars,

**@RequestParam** - Get – ի միջոցով փոխանցված պարրամետրերի մշակումը Controller – ում կարող է կատարվել HttpServetRequest օբյեկտի getParameter(“name”) մեթոդի կամ **@RequestParam(“name”)** անոտացիայի (եթե փոփոխականի անունը նույնն է ինչ որ պարամետրինը առանց արգումենտի) 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): Եթե պարամետրի փոփոխականը Map է ապա կունենանք պարմետրերի և դրանց արժեքների key – value map:

**@RequestHeader –** @RequestParam - ի սկզբունքով request header – ների արժեքները արգումենտի փոփոխականներին վերագրելու համար:

**@CookieValue –** cookie արժեքները փոփոխականին վերագրելու համար:

**@ModelAttribute –** Model օբյեկտի ատրիբուտների հասանելության համար

**@SessionAttribute, @RequestAttribute –** սեսիայում, request - ում գոյություն ունեցող ատրիբուտը ստանալու համար, ընդ որում կարող են լինել նաև բարդ տիպեր օր՝ @SessionAttribute User user

**@RequestBody -** Օգտագործվում է post – ի ժամանակ։ Կլիենտի ուղարկած json – ը ավտոմատ վերածվում է օբյեկտի jackson – ի միջոցով և վերագրվում անոտացիայով նշված օբյեկտին։ Որպեսզի jackson – ը կարողանա parse անել json – ը օբյեկտի, օբյեկտի կլասսը պարտադիր պետք է ունենա կամ դատարկ կոնստրուկտոր կամ պարամետրավորված կոնստրուկտոր նշված @JsonCreator անոտացիայով, իսկ պարամետրի փոփոխականները պետք է ունենա @JsonProperty(“properyName”) անոտացիան: HttpEntity<Object> օբյեկտը արգումենտում կատարում է նույն ֆունկցիան ինչ որ RequesBody – ն այն տարբերությամբ որ HttpEntity – ն իր մեջ պարունակում է body – ից բացի նաև header – ները:

**Exceptions**

@Controller կլասսները կարող են ունենալ **@ExceptionHandler** մեթոդներ միայն տվյալ կլասսի controller մեթոդներում ի հայտ եկած exception – ները մշակելու համար: **@ExceptionHandler ({IOException.class})** պարամետրում նշվում է exception – ի կլասսը՝

@ExceptionHandler({FileSystemException.class, RemoteException.class})

public ResponseEntity<String> handle(IOException ex) {

// ...

}

Սովորաբար օգտագործվում է **@ControllerAdvice(basePackages = “endpointsPackagePath”) կամ @ControllerAdvice(annotations = RestController.class)** անոտացիայով կլասսը որի մեջ հայտարարվում են բոլոր @Exceptionhandler անոտացիայով մեթոդները controller կլասսների խմբի համար կախված package – ից կամ կլասսի վրա կիրառված անոտացիայից:

1-ից ավելի Exception մեթոդներից կաշխատի ամենանեղ exception – ով մեթոդը:

**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։

**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 աշխատում 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

**postForEntity** և **getForEntity** մեթոդները վերադարձնում են ResponseEntity<T> օբյեկտ, այսինքն body – ն և header – ները:

**ResponseEntity<T> exchange** – մեթոդը request – ի ունիվերսալ տարբերակ է որը կարող է ընդունել request մեթոդի տեսակը, HttpEntity և այլն:

**T execute** – նույնը բայց վերադարձնում է T տիպի օբյեկտ

Մեթոդները որպես պարամետր ընդունում են uri շաբլոն որը իր մեջ պարունակում է path variable փոփոխականներ, իսկ դրանց վերագրումները սովորաբար կատարվում է map – ի միջոցով որը նշվում է որպես մեթոդի վերջին արգումենտ օր՝

Map<String, String> vars = Collections.singletonMap("hotel", "42");

String result = restTemplate.getForObject(

"https://example.com/hotels/{hotel}/rooms/{hotel}", String.class, vars);

Request պարամետրներ փոխանցելու համար՝

UriComponentBuilder builder = UriComponentBuilder.fromHttpUrl(String url)

builder.queryParam(“param1”, value1)

builder.queryParam(“param2”, value2)

builder.toUriString()

**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։

**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:

**UriComponents**

UriComponentBulder – ի միջոցով ստեղծվում են URI – ներ շաբլոնների օգտագործմամբ: Օր՝

UriComponents uriComponents = UriComponentsBuilder

.fromUriString("https://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.encode()

.build();

URI uri = uriComponents.expand("Westin", "123").toUri();

Կամ ավելի կրճատ՝

URI uri = UriComponentsBuilder

.fromUriString("https://example.com/hotels/{hotel}")

.queryParam("q", "{q}")

.build("Westin", "123");

Կարելի է ստանալ նաև որևէ mapping – ի հղումը controller կլասսից հետևյալ կերպ՝ ( @GetMapping("/bookings/{booking}") getBooking (Long value) - համար)

UriComponents uriComponents = MvcUriComponentsBuilder

.fromMethodName(BookingController.class, "getBooking",21).buildAndExpand(42);

URI uri = uriComponents.encode().toUri();

21 – ը մեթոդին փախանցվող արգումենտն է իսկ 42 և ավելի pathvariable – ների արժեքները: