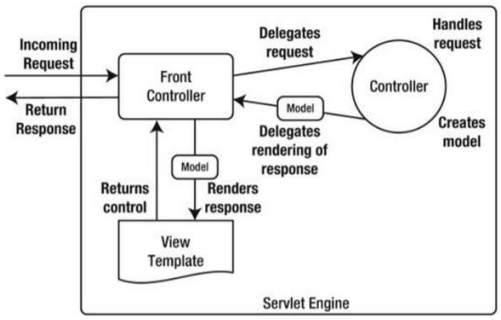
Spring MVC

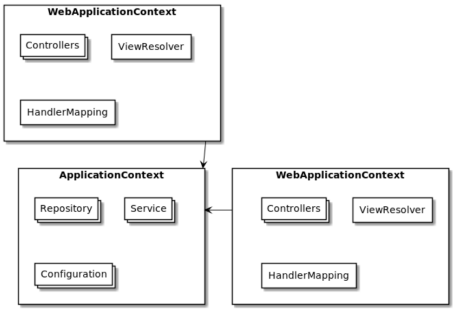
Spring MVC- модул изграден врз JEE Servlet API и е вклучен од почеток на Spring рамката. Тој е дизајниран да изработува HTTP барања со централен сервлет **DispacherServlet** кој пренесува барања до контролерите и нуди дополнителни функционалности кои олеснуваат при развој на веб апликации. Овој сервлет исто се нарекува и **Front Controller** а и е дел од **Presentation Layer** плус е одговорен за наоѓање на погледот (View). Сервлетот ги пренасочува барањата кон RequestHandler методи т.е. методи со анотација RequestMapping.



Мапирањето на методи за справување со барања (**HandlerMapping**) е начин на конфигурирање на поврзувањето на методите за справување со барања со параметрите на HTTP барањето.

MethodArgumentResolver - механизам за разрешување на аргументите на методите и овозможува да се вметнат аргументите на методите за да може да се справиме со барањата врз основа на анотациите (@RequestParam, @RequestHeader, @PathVariable...).

Пресретнувачи (**HandlerInterceptor**) се потребни кога ни се потребни посебни процесирања по справување со барање. Може да се користи за кеширање и се додаваат headers ('Cache-Control','Pragma','Expires').



DispacherServlet користи сопствен контекст (**WebApplicationContext**) кој е проширување на главниот контекст на Spring (**ApplicationContext**). И ако имаме повеќе DispacherServlet- и тогаш ќе имаме и повеќе WebApplicationContext- и.

HandlerMapping - конфигурација за мапирањето на HTTP барања до методите за справување со овие барања како на пр за @RequestMapping методи: (@GetMapping, @PostMapping, @DeleteMapping, @PutMapping, @PatchMapping ...).

HandlerAdapter - помагаат на DispacherServlet-от да ги повика методите за справување на тие барања без потреба на тој да ги знае нивните детали. Се извршува кога прво ќе се најде HandlerMapping методот па потоа ги разрешува аргументите (креира Model) и се повикува handle методот. **HandlerExceptionResolver** е за справување со исклучоци.

Во Spring има две опции на дефинирање контролери: со ***анотации***, со ***имплементација на интефејс*** и истиот треба да се имплементира е org.springframework.web.servlet.mvc.Controller и најчесто е комбиниран со SimpleUrlHandlerMapping така што сите URL патеки се на една локација.

ViewResolver - конфигурација за справување со прегледите. За разрешување на логичките имиња на прегледи во **вистинскиот преглед (View)** со кој ќе се генерира изгледот на страната на response-от. Кога метод од контролер враќа име на HMTL фајл тој тогаш генерира преглед и ги зема атрибутите од **Model** објектот. Може Се додаваат суфикс и префикс за бараниот преглед во application.properties фајлот со spring.thymeleaf.prefix и spring.thymeleaf.suffix

LocaleResolver, **LocaleContextResolver** - за разрешување на јазикот што го користи клиентот и неговата временска зона за да може да се креира интернационални прегледи (Views).

MultipartResolver -апстракција за справување на барања кои се составени од повеќе делови. За справување на датотеки кои се прикачени преку форма на browser (file upload).

RequestMapping подржува својства на совпаѓање со барањето по:
-**URL патека** преку path или value -> @RequestMapping (path = {"", "/products"})
-**HTTP метод** преку method -> @RequestMapping (method = {RequestMethod.PUT, RequestMethod.PATCH})
-**Параметри** преку param -> @RequestMapping(param="myParam=myValue")
-**Заглавја** преку header -> @RequestMapping (header = "myHeader!!=myValue").
-**Типови на содржини испратени** преку consumes -> @RequestMapping ("/something", consumes = "text/*")
-**Типови на содржини прифатени** преку produces -> @RequestMapping ("/something", produces = "text/*")

PathPattern ги процесира URL патеките по следниве правила:
? - заменува еден знак
* - заменува нула или повеќе знаци во сегментот. Сегмент е делот меѓу две коси црти ("/segment/")

** - заменува нула или повеќе сегменти. Не е дозволена на средина да стои ("/**/{id}") или ("/**.txt")
{var_b}- одговара на сегмент и го зема како променлива во патеката path variable со име "var_b".
{var_b: [a-z] +} - одговара на сегмент но ако сегментот каде што е var_b го има регуларен израз [a-z] +
{*var_b} - заменува нула или повеќе сегменти и го зема како променлива последниот сегмент.

Handler Method- @PathVariable - за шаблони на URL патеки кои имаат променливи. Ако нема наведено променлива во патеката но се бара во методот тогаш се фрла HTTP статус 500 (Internal Server Error).

Handler Method- @RequestParam - кога бараме параметар во барањето. Ако не е присутно а е задолжително тогаш HTTP статус 400 (Bad Request). Може да го направиме опционално со @RequestParam (required = false) или да е од класа Optional<>. Со Map<String, String> или MultiValueMap<String, String> можеме да ги пристапиме сите параметри од барањето.

Handler Method- @RequestHeader - кога бараме заглавје од барањето, исто однесување како @RequestParam.

Handler Method- @CookieValue - кога бараме колачиња од барањето, исто однесување како @RequestParam.

На овие сите Handler-Methods наведени горе може да бараме аргументи од различни типови но тие се пратени во тип string.Па има 3 начини да тие се конвертираат. Со:
- **Конвертори** (**Converters**)- претвораат од еден јава тип во друг.
- **Форматери** (**Formatters**)- претвора на String во друг јава тип и обратно.
- **Уредувачи на својства** (**Property editors**)- стариот начин на конвертирање на својства.

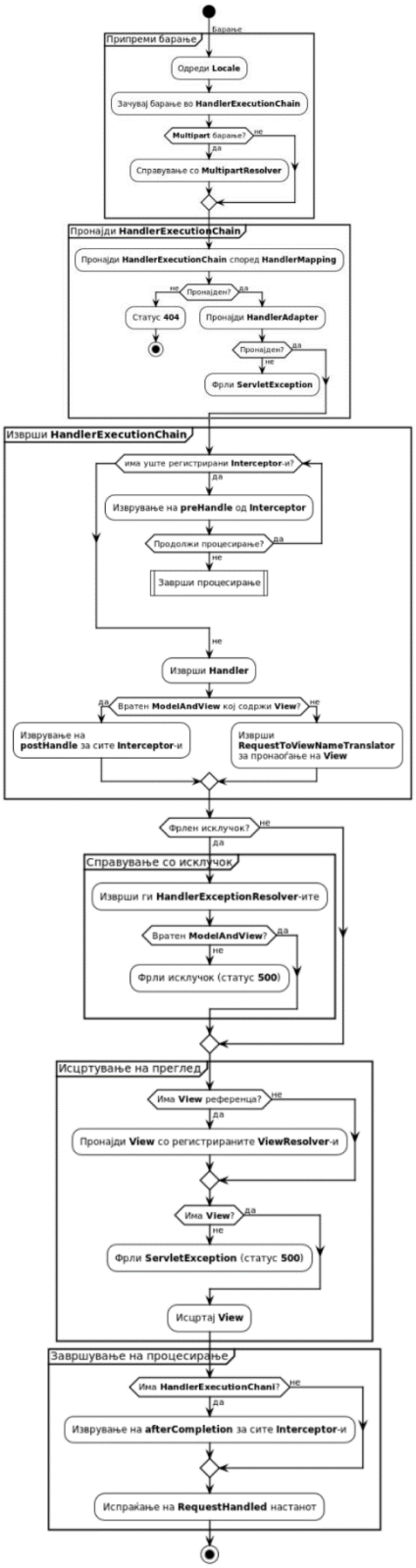
Останати Handler-Methods се: **ServletRequest**, **ServletResponse**, **HttpSession**, **HttpMethod**, **Locale**, **TimeZone**, **ZoneId**, **InputStream**, **SessionAttribute**, **RequestAttribute**, **Reader**, **OutputStream**, ,**Multipart**/ **MultipartFile**/ List<MultipartFile> / Map<String, MultipartFile> (датотеки кои се прикачуваат,) **Writer**, **Model**, **RequestBody** (парсира телото во објект,се користи content-type за да се одради начинот на прасирање), **HttpEntity** (ПОгенерална форма на RequestBody), **ModelMap**, **RequestPart** (за пристап на дел од барањето со тип содржина за пристап до дел од барањето со тип на содржина multipart/form data), **ModelAttribute**(се анотира на ниво на метод,го зема тоа што е во return, или ако е во метода тогаш тоа по него), **SessionAttributes** (складира атрибути на моделот во HTTP сесијата,на ниво на класа се анотира,останува во сесијата додека не се повика SessionStatus.setComplete())

Подржани повратни вредности на методите на контролерите се:
- **ModelAndView** - експлицитно се дефинира приказот,моделот и статусот на одговорот.
- **View** - експлицитно се дефинира приказот
- **Map/Model** - атрибутите треба да се додадат како атрибути на моделот за прикажување, и за преглед се зема името на методот.
- **ModelAttribute** - атрибутот треба да се додаде како атрибут на моделот за прикажување, и за преглед се зема името на методот.
- **String**- за преглед се зема резултатот на методот
- **void** - се смета дека со void или null методот целосно се справил и нема потреба за одговор.Може да значи и 'одговор без тело' за REST контролери.
- **ResponseBody** - резултатот се сместува директ во одговорот ,ако не е наведено produces својството, тогаш резултатот ќе се конвертира во JSON одговор.
- **HttpEntity<B>, ResponseEntity<B>** - слично како ResponseBody но со статус и заглавја.
- **HttpHeaders** - враќа одговор со заглавја но БЕЗ тело.

Во досегашното градење на веб апликации ние работевме со тоа клиент праќа барање,сервер процесира и враќа одговор да се прикаже. Но во **REST (Representational State Transfer)** ние не враќаме приказ туку враќаме JavaScript или JSON/XML.И со тоа после стануваат достапни за AJAX.REST интерфејсите APIs користат униформни идентификатори на ресурси URI за адресирање на ресурсите. Методи кои се користат во REST архитектурниот шаблон се: **GET**, **POST**, **DELETE**, **PUT** (заменува цел податок), **PATCH** (прави ажурирање на податокот), **OPTIONS**. Најчести статус кодови се: **200-OK**, **201-CREATED**, **204-NO CONTENT** (успешно избришан), **304-NOT MODIIFED** (најчесто користен за кеширање за дали ресурсот е променет), **400-BAD REQUEST**, **401-UNAUTHORISED**, **403-FORBIDDEN**, **404-NOT FOUND**, **500-INTERNAL SERVER ERROR**.

Страничење е потребно затоа што не знаеме колкава е количината на податоци,исто и филтрирање.

Spring Application може да конзумира REST API со RestTemplate(едноставен,синхронизиран REST клиент обезбеден од Spring). RestTemplate обезбедува 41 методи за интеракција со REST ресурси и уникатните 12 се:
-**exchange(...)** -извршува HTTP метод на URL враќа ResponseEntity што има објект од телото на одговорот.
-**execute(...)** - извршува HTTP метод на URL враќа објект од телото на одговорот.
-**getForEntity(...)** - праќа GET барање за ResponseEntity што има објект од тело
-**getForObject(...)** - праќа GET барање за да врати објект од тело
-**headForHeaders(...)** - праќа HEAD барање да врати заглавија од HTTP на наведена URL.
-**optionsForAllow(...)** -праќа OPTIONS барање да врати заглавјето Allow за наведената URL.
-**patchForObject(...)** - праќа PATCH барање, враќа објект од телото на
-**postForEntity(...)** -праќа data по POST барање и враќа ResponseEntity што има објект од телото
-**postForLocation(...)** -праќа data по POST барање и враќа URL на новиот објект
-**postForObject(...)** -праќа data по POST барање и враќа го добиениот објект од телото
-**put(...)** - праќа data по POST барање на URL
-**delete(...)** -извршува DELETE барање на ресурс на URL



Работа со Бази на Податоци – Java Persistence API (JPA)

Кога имаме потреба перманентно да чуваме податоци, ние треба да ги чуваме тие во база на податоци. Најчесто се користат релационите бази на податоци. Но предизвик е тоа што во база се чуваат табели а во апликациите се објекти.

Чекори за пребарување податоци во база преку апликација се: постави конекција, испрати SQL барање за читање data , преземи резултат , convert на податоци од табеларна форма во објект, затвори конекција и справи со грешки на секој чекор.

Овие сите чекори се овозможени со имплементација на **Java Database Connectivity (JDBC)** - стандарден API за комуникација со релациони бази на податоци и содржи драјвери за сите типови. Но кодот е комплексен поради тоа што треба на секое барање да се извршат сите чекори. Па затоа ни е потребна рамка за поедноставување не сето ова. Тие рамки треба да го прават следното:

- **ORM (Object Relational Mapping)** – пресликување на објект во табела.
- **Механизам** за размена на објекти помеѓу апликациите и базите.
- **Меѓусебна компатибилност** и нецелосна транспарентност.

Во јава имаме рамка за ова и тоа е **JPA(Java Persistence API)** која е дел од JEE. Таа е **спецификација** од типот <u>отворен код која дефинира само правила **но не имплементација**</u>. Делови на JPA се: **ORM**,**CRUD**, **JPQL**(java persistence query Language)- за пишување напредни пребарувања.

Имплементација на JPA ја прават провајдерите Hibernate, EclipseLink кои генерираат SQL наредби кои се извршуваат преку JDBC. И овие имплементации ја кријат комплексноста, но за секоја мапирана класа треба да се креираат репозиториум со ООП на **CRUD** операции а со ова ќе се повторува код и помала е ефикасноста на програмерот. Ова може да се реши со **Spring Data JPA** која нуди апстракција за работа со секоја база на податоци. Апстрактни функции се: **CRUD** операции дури и кога нашата имплементација на репозиториум не ги содржи(**празен репозиториум**)**,** креира имплементации на посложени пребарувања преку гледајќи го **името на методата во репозиториумот.**

На кратко JDBC дава **апстракција на конекција и пишување код за База**, JPA имплементација(Hibernate или EclipseLink) даваат апстракција за **ORM,CRUD,JPQL ,**и Spring Data дава апстракција за имплементации на **CRUD операциите и имплементации на сложени пребарувања**.

ORM (Object Relational Mapping) – пресликува објекти во табела и обратно. Објект се обележува преку анотации или во .xml датотека. **Ентитет** е објектот кој се пресликува во табела и секоја инстанца е посебен ред во табела а пак својствата се колони. Својства на ентитет се:

- **синхронизирани**(секоја промена на инстанца исто да се преслика и во база),
- **уникатност**(секоја инстанца уникатна во нешто(**примарен клуч**)),
- **трансакционалност**(при грешка при синхронизација промената нема да се случи).

Секоја класа за да биде ентитет треба да е анотирана со **@Entity**, да има колона **@id**,и на има **празен конструктор** (затоа што JPA прво креира инстанца со анализирање, па потоа со сетери ги полни колоните). Именувањето на табелите и колоните **зависи од JPA имплементацијата**.

Пресликување на примарните клучеви со **@id** не генерира сама по себе, па потребно е **@GeneratedValue** и со аргумент **strategy** со можни вредности:

- → **AUTO** - (default и автоматски користи еден од следните три методи),
- → **SEQUENCE** - (increment со глобален бројач, последна вредност се чува директно во база во објект sequence која генерира низа уникатни вредности),
- → **IDENTITY** - (increment со приватен бројач),
- → **TABLE** - (increment со глобален бројач, но гледа која е последната бројка во специјална табела).

JPA ги подржува **сите примитивни типови**, **листи**, **колекции**, **мапи**, **Date**, **Time**, **енумерации**, **серијализибилни класи (JSON)**. Но енумерациите за да станат трајни треба да се анотираат со **@Enumerated** и (default) однесување е **EnumType.ORDINAL(**зачувува енум како бројка "0", проблем ако потоа се додаде нов енум**),** а со **EnumType.STRING(**вредноста на енум се зачувува директно како стринг "ACTIVE"**).**

Транзитни полиња се тие кои се игнорираат и не се додаваат како колона во база. Пр. пресметани вредности или temp вредности. Се анотира со **@Transient**.

Вградени објекти- оние кои не постојат самостојно(нема редица во колоната) ако не се како својство на некој друг ентитет и немаат свој идентитет (примарен клуч), т.е. мора да имаат сопственик/ци. Класата се анотира со **@Embeddable**, а својството на родителот со **@Embedded**

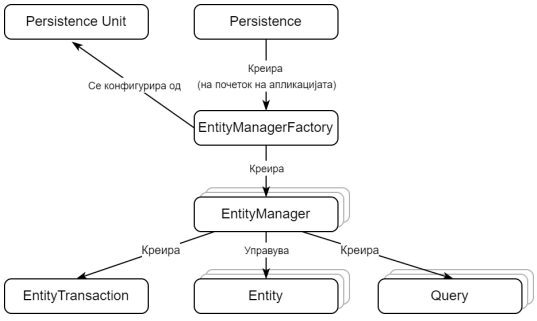
Релациите меѓу ентитети може да бидат: **еднонасочни** , **двонасочни** .Типови на релации се:

- **one-to-one** – дестинацијата не може да биде во релација со друг. За двонасочна се користи mappedby во дестинација која кажува името на својството во класата на сопственик. Се анотира со **@OneToOne**.
- **one-to-many** - изворот може да е поврзан со повеќе дестинации. Се користи колекција. Ако е еднонасочна тогаш JPA не знае каде да го смести надворешниот клуч, па со **@JoinColumn** наведуваме каде да се смести клучот. Својството има тип: **List<>**, **Set<>**, **Map<>.** Се анотира со **@OneToMany**.
- **many-to-one** – дестинацијата има повеќе извори. Најчесто се мапира клучот кај изворот. За двонасочна тогаш се додава @OneToMany кај изворот за и дестинацијата да го добие клучот. Се анотира со **@ManyToOne.**
- **many-to-many** - дестинација може да има повеќе извори а и изворот може да има повеќе дестинации. Се користи колекција. Својството има тип: **List<>**, **Set<>**, **Map<>.**Се креира нова табела. За да биде двонасочна тогаш треба и кај другиот ентитет да ја има анотацијата.Ако сакаме да ја именуваме оваа нова табела тогаш користиме **@JoinTable(name="new_name")**,а може да ги смениме и имињата на колоните во неа со додавање **@JoinTable(name="t1", joinColumns = @JoinColumn(name = "prod "), inverseJoinColumns = @JoinColumn(name = "cat "))**.

Ако кај релацијата се стави **@relationAnnotation(mappedby='foreign entity attribute name')**,тогаш клучот ќе биде во дестинацискиот ентитет. А ако се стави **@JoinColumn(name='column_name')** тогаш истиот ентитет кој го има наведено ова ќе го има клучот. Ако сакаме да ги сортираме низите кај релации, тогаш се користи **@OrderBy("column [ASC/DESC]")**.

Кога се креира инстанца од ентитет тоа не значи дека е синхронизирана со базата. За да може да се управува со ентитети тие мора да станат перзистентни.

EntityManager е JPA интерфејс кој е одговорен и управува со ентитети. Кога се прави операција кон база оваа класа креира објект **EntityTransaction** (се отвора и се затвора - **singleton**) кој овозможува атомично извршување.



Query е исто објект кој го креира кога корисникот зака да изврши кориснички-дефинирано пребарување. Секој EntityManager е креиран(на секое барање) и управуван од еден објект **EntityManagerFactory** кој се креира од **Persistence** при старт на апликацијата според конфигурацијата во persistence.xml. Фабриката креира и управува со JDBC конекции и ги доделува на EntityManager's. EntityManager може да биде управуван од:

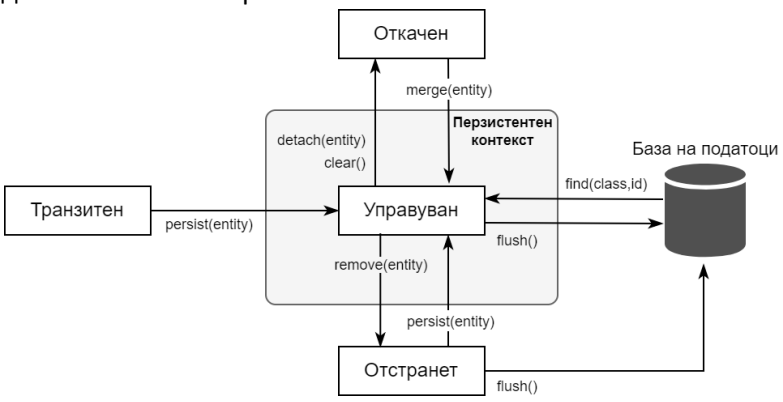
- **апликацијата(application managed)-**програмерот 1.креира инстанца од EntityManagerFactory, 2.креира **инстанца од EntityManager**, 3.управува со **животот на EntityManager**, 4.креира објект од **EntityTransaction**, 5.да започне трансакција со **EntityTransaction.begin()**, 6.повика методи од **EntityManager за работа со ентитети**, 7.синхронизирај промени со **EntityTransaction.commit(),** 8.затвори го EntityManager со **EntityTransaction.close()**. Ова има контрола но е покомплексно
- **контејнер(container managed)**- од контејнер(пр.**Spring**) е управуван и програмерот треба само да добие инстанца од EntityManager со **@PersistanceContext** и да управува со него.

Ако метод работи со ентитети преку EntityManager, тој се означува со **@Transactional**. Со (Aspect Oriented Programing) контејнерот креира код (пред да влезе во методот креира трансакциски објект и го отвара за работа, а кога ќе излезе од методот ако нема други вакви методи, прави **commit** и затвара трансакцијата).

Persistence Unit е перзистентна единица која ги содржи информациите за: **базата на податоци** (параметри за најава), **JPA**, **конфигурациите на ентитетите**. Ако имаме потреба од повеќе бази во исто време да користи апликацијата тогаш **може да има повеќе Persistence Unit**.

Секој EntityManager управува со ентитети кои се во **перзистентниот контекст**(мемориска единица која ги содржи перзистентните објекти). Било која промена во овој контекст се ажурира во база. Ако инстанца не е во контекстот тогаш таа не ажурирана во база. EntityManager ги има следниве операции за ентитет:

- **persist(entity)**-додавање на ново-креирани ентитети во перзистентниот контекст преку методот
- **remove(entity)**–бришење на ентитет од перзистентниот контекст и маркира за бришење.
- **detach(entity)** – отстранување на **entity** од перзистентен контекст и промени не се синхронизираат со база.
- **merge(entity)**–додавање на постоечки ентитет **entity** во перзистентниот контекст. Го враќа ако бил **detached.**
- **refresh(entity)** – присилно го ажурира **entity** со податоци од базата на податоци.
- **find()**-бара инстанца во база со класа **Entity** со индекс **id** во базата. Враќа **null** ако нема.
- **flush()**-зачувување(**forced synchronization/save**) на сите промени во ентитетите во перзистентниот контекст во базата на податоци
- **clear()**-откачување на сите ентитети од перзистентниот контекст
- **contains(entityInstance)** – бара инстанца од ентитет **entity** во перзистентниот контекст.
- **createQuery()-**извршување на уникатни пребарувања во перзистентниот контекст



JPA: Животен циклус на Ентитет:

- **Транзитен** (Transient) - ентитетот не е во перзистентниот контекст бидејќи е ***new***. А може да се внесе со **persist(entity)**.
- **Управуван** (Managed) – ентитетот е во перзистентниот контекст и е управуван од **Entity Manager** и било која промена на ентитетот ќе се синхронизира и со базата на податоци.
- **Откачен** (Detached) – ентитетот бил во перзистентниот контекст но е отстранет од **Entity Manager** со метод ***remove***
- **Отстранет** (Removed) – ентитетот е отстранет од контекстот и е означен за бришење. И ако не се врати назад во контекстот тогаш таа редица/објект ќе биде отстранет од базата на податоци.

Кога ентитет се креира со конструктор тој е во **транзитна** состојба и не е во перзистентниот контекст. За да стане управуван и синхронизиран со база тој треба да се внесе со **persist** методата. За да се вчита постоечки објект од база тоа се прави со **find(Entity.class, id)** и ако врати инстанца тогаш ентитетот е најден ако враќа null тогаш не е најден во база. А исто ова ќе а стави најдената инстанца во контекстот и ќе стане ***управуван***. Класата на методот помага при конверзија во објект ако инстанца се најде. Ако сакаме да најдеме ентитет во перзистентниот контекст тоа може со **contains(entityInstance)** каде се бара инстанцата **entityInstance** во контекстот и ако е во него враќа **true** ама ако е отстранет или откачен тогаш враќа **false**. Ако сакаме присилно да синхронизираме инстанцата **entity** од

информациите од база тогаш тоа со **refresh(entity)**. Ако инстанца е пратен до frontend за промена и тој станува ***откачен*** со **detach(entity)** и не е ***управуван*** а со **merge(entityInstance)** таа инстанца ќе стане ***управувана*** и ќе биде потоа синхронизирана во база. Ако сакаме да избришеме ентитет тој прво треба да се стави од база во контекстот и да стане ***управуван*** а потоа со **remove(entity)** ентитетот ќе биде означен за бришење а истото се извршува по затворање на трансакцијата или пат да се врати во контекстот со **persist(entity)**. Ако сакаме присилно да ги зачуваме (**forced synchronization/save**) сите ентитети од контекстот во база тоа може со **flush()**. Ако сакаме сите инстанци да ги откачиме од контекстот тоа можеме со **clear()** а ако сакавме специфична инстанца да откачиме тоа со **detach().**

Ние можеме да извршиме и кориснички дефинирани методи за работа со JPA тоа го можеме со JPQL(Java Persistent Query Language) или со рачно напишан SQL. Тоа може со класите:

- **TypedQuery** – за извршување на динамички пребарување дефинирани со JPQL
- **NamedQuery** – за извршување на статички пребарувања дефинирани со претходно зачувани JPQL.
- **NativeQuery** – за извршување на динамички пребарувања дефинирани со SQL

Сите овие класи ги имаат методите **getResultList()**, **getSingleResult()**, **executeUpdate()** -> извршува барање за ажурирање или бришење и не враќа конкретен резултат.

Ако ентитетите имаат релации тогаш при читање на нив се читаат и нивните релации. А тие дестинациски ентитети во релацијата се добиваат со ***get*** метода . Има два начини на однесување при читање на релациите. Тоа се:

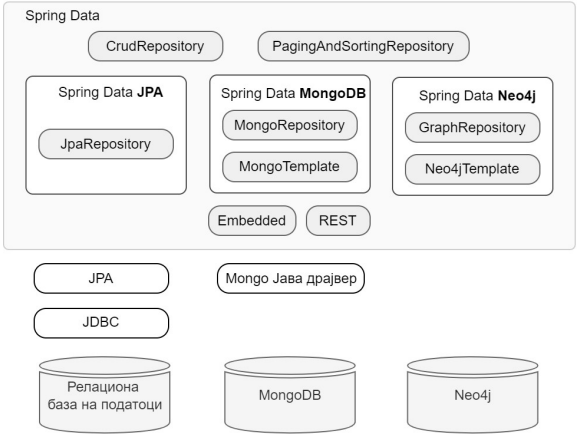
- **LAZY –** мрзливо однесување каде дестинациските ентитети не се читаат при читање на изворниот ентитет. Па за читање на тие ентитети треба посебно да се повика метод за читање. Најчесто кај **1:M** и **M:N.** Предифинирано однесување кај **(1/M) *:N .** Се анотира со **fetch=FetchType.LAZY**
- **EAGER –** дестинациските ентитети се читаат при читање на изворниот ентитет. Предифинирано однесување кај **(1/M) *:1** врски. Се анотира со **fetch=FetchType.EAGER.** Ќе резултира лоши перформанси кај **(1/M) *:N**

Уредување на ентитети во релација каде при додавање или отстранување на ентитет ,потребно е да се дефинира како ќе се однесува релацијата. Ова е важно затоа што ако некој ентитет е во релација и му се изврши промена тогаш треба и релацијата да се ажурира. Тоа се прави со дефинирање на методот за пренесување на состојба на изворниот со дестинацискиот ентитет кој се нарекува **cascade(каскада)**. И има повеќе вида на каскади како:

- **PERSIST –** повикот **persist** на изворниот се пренесува и кај дестинацискиот ентитет. **cascade=CascadeType.PERSIST**
- **MERGE -** повикот **merge** на изворниот се пренесува и кај дестинацискиот ентитет. **cascade=CascadeType.MERGE**
- **DETACH -** повикот **detach** на изворниот се пренесува и кај дестинацискиот ентитет. **cascade=CascadeType.DETACH**
- **REMOVE -** повикот **remove** на изворниот се пренесува и кај дестинацискиот. **cascade=CascadeType.REMOVE**
- **REFRESH -** повикот **refresh** на изворниот се пренесува и кај дестинацискиот. **cascade=CascadeType.REFRESH**
- **ALL** – било кој повик на метод од горе-наведените ќе се пренеси и кај дестинацискиот. **cascade=CascadeType.ALL**

Но треба да се пази при користење на **CascadeType.ALL** затоа што во него е и **REMOVE** кој може да направи загуба ако изворниот ентитет се избрише а со тоа и сите дестинациски ,па тоа е проблем ако не го сакаме тоа. Но ако се случи бришење на дестинациски од **1:*(M/1)** тогаш дестинацискиот не се бриши од база само во релацијата наместо неговиот индекс вредноста ќе стане **null**. А ако ентитет не е веќе референциран од ниеден изворен ентитет тогаш тој станува **сирак (orphan)**. Ние можеме да поставиме однесување за orphans со **orphanRemoval** каде ако вредноста е **true** тогаш ако ентитет стане сирак тогаш тој ќе биде отстранет, но ако е **false** тогаш нема да се отстрануваат сираци. **orphanRemoval** е за **1:*(M/1) релации** а предефинирана вредност е **false.** Каскадните промени(промена на вредност или состојба[TRANSIENT, MANAGED..]) се **одвиваат после промена на изворниот ентитет. Но прво се креира секвентен број(sequence number) за индекс на секој нов ентитет при додавање**.

Spring Data - За CRUD работа со ентитети ќе треба да се генерираат различни **репозиториуми со сличен код** а ова доведува до дуплирање код, помала продуктивност, поголема можност за грешка. Ова може да се среди со **Spring Data** кој е проект на **Spring** за работа со податочниот слој кој не зависи од типот на база. И Spring Data автоматски генерира код за CRUD операции преку **интерфејси за репозиториуми**.



Repository е највисокото ниво на апстракција со кој кажува дека Spring Data е задолжен за овозможување на имплементација на интерфејсот. Служи само за маркирање на типот на податоци.

CrudRepository – нуди имплементација на CRUD и останати операции врз објектите. Работи за било кој тип на база. Ги има методите **save**, **findById**, **findAll**, **count**, **delete**, **existsById**. За дефинирање на репозиториум за работа со некој ентитет потребно е тој репозиториум да го проширува (**extends**) CrudRepository.

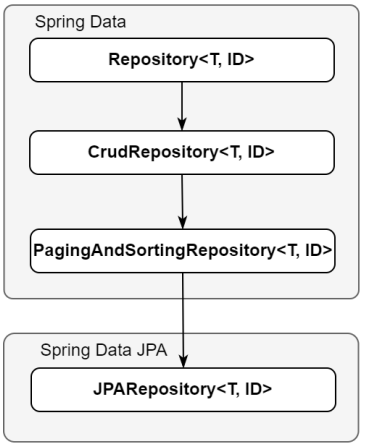
PagingAndSortingRepository – кој нуди имплементација на методи за страничење и подредување и го проширува (**extends**) CrudRepository од кој ги наследува и неговите методи. Методи на **PagingAndSortingRepository** се:

- **findAll(Sort sort)** – враќа подредена листа од ентитети со подредување дефинирано преку објектот **sort**. Пример за тоа е Sort.Direction.ASC или Sort.Direction.DESC или пак Sort.by(Sort.Direction.ASC,"date").
- **findAll(Pageable pageable)** -враќа објект од тип **Page** кој содржи дел од целата колекција на ентитети. А методи на објектот се:
  - **getContent()** – враќа колекција од ентитети кои се дел од таа страница
  - **getTotalElements()** – го враќа цел број на ставки(ентитети) од (сите страници)
  - **getTotalPages()** – го враќа вкупниот број на страници.

  Објектот **Page** овозможува дефинирање на : големина на страница, бројот на страница која треба да се врати, подредување преку назначување на објектот од тип **Sort.** Пр: findall(1,10, Sort.by(Sort.Direction.ASC,"date")) кое ќе врати страницата со индекс 1 и има 10 ставки во неа и ги има ставките од 10-19 подредени по "date"

JpaRepository – го проширува (**extends**) PagingAndSortingRepository и има методи специфични за JPA. Методи се:

- **findAll()** – враќа листа од објекти **List.**
- **flush()** – повикува **flush()** за моментална синхронизација од контекстот кон база**.**
- **saveAll(Iterable<S> entities)** – зачувува листа од ентитети**,**
- **saveAndFlush(S)** - повикува **save()** и **flush()** за ентитетот инстантно да се зачува во база.
- **saveAllAndFlush(Iterable<S> entities)** - инстантно зачувува листа од ентитети во база**.**
- **DeleteInBatch(Iterable<S> entities)** – брише листа од ентитети



Spring Data овозможува креирање на специфични барања за ентитети само преку дефиниција на името на методот. Пр: findByCustomerId(Integer id) . И конвенција за именување методи се: **findBy**, **readBy**, **getBy**, **searchBy, queryBy** враќаат листа од ентитети кои го задоволуваат условот кој следи во името. **countBy** враќа број на ентитети кои го задоволуваат условот во името. **deleteBy** и **removeBy** бришат ентитети кои задоволуваат условот во името. **existsBy** враќа true ако постои ентитет кој го задоволува условот. **...First<number>…** и **…Top<number>…** се стават меѓу **find** и **By** за да се ограничи бројот на ставки во резултатот (пр. findTop10ByName). **…Distinct…** се става меѓу **find** и **By** за да се врати само уникатните ентитети (пр. findDistinctByName).

А после дефинирањето на типот на пребарување следуваат **предикати кои ги дефинираат условите**. И секој предикат започнува со голема буква и се третира како атрибут се до појавување на клучен збор или EOL . А ако постојат повеќе зборови со голема буква тогаш сите се третираат како својство со истото име но со мала буква. Пр: findByCustomerId(Integer id) -прво проверува постоење на својство customerid ако не тогаш customer.id . Ако експлицитно пребарување по својство кој е дел од сложено(composite) својство тогаш се користи знакот '_' долна црта. На секое својство во условот одговара **аргумент од истиот тип** чија позиција зависи од **позицијата на својството во името**. А ако станува збор за бинарен оператор (пр: between) тогаш се користат два аргументи.

Сега ако имаме потреба за сами да пишуваме барања(прашалници) во SQL кои неможат да се имплементираат преку има на методата тоа може да се направи со наши дефинирани **барања со користење на JPQL или SQL** so користење на анотацијата **@Query** над методата. Сега ако аргументот ***nativeQuery=false*** барањето се третира како JPQL (предефинирано), а ако е ***nativeQuery= true*** тогаш барањето се третира како SQL. Тука се игнорира името на методата. Овие барања се дефинираат во репозиториум кој го наследува **JpaRepository.** Означување на параметрите во барањето е преку :

- **Ime:** во барањето (SQL question), а **@Param("ime")** се става пред аргументот во методата.

```
@Query("select o from Order o where o.customer.id = :id and o.status =:status ")
public List<Order> findByCustomerIdAndStatus(@Param("id") Integer customerId, @Param("status") OrderStatus satus);
```

- **?n** во барањето (SQL question), каде **n** е позицијата(индекс) на аргументот во методот.

```
@Query("select o from Order o where o.customer.id = ?1 and o.status = ?2 ")
public List<Order> findByCustomerIdAndStatus(Integer customerId,OrderStatus status);}
```

- Напредно барање со страничење **Pageable**

```
@Query("select o from Order o where o.customer.id = ?1 and o.status = ?2 ")
public Page<Order> findByCustomerIdAndStatus(Integer customerId, OrderStatus status, Pageable pageable);}
```

- Кориснички дефинирани барања кои прават промена во база (Update, Create, Delete) со анотација **@Modifying**

```
@Modifying
@Query("update Order o set o.amount = (1-:discount*1.0/100)*o.amount where o.customer.id = :id ")
public List<Order> updateCustomerIdAmount(@Param("id") Integer customerId,
 @Param("discount") Integer discount);}
```

Spring Data JPA нуди можности за напредни функционалности како (надвор од целта на книгата):

- **Складирани процедури (Stored Procedures)** – процедури напишани во SQL и се креирани и зачувани во база.
- **Спецификации(Specifications)** – овозможува динамичко градење на пребарување додавајќи предикати во пребарувањето
- **Ревизија(Auditing)-** водење на дневник за модификација на ентитетите кој вклучува кориснички и временски податоци за модификацијата
- **Именувани ентитетски графови (Named Entity Graphs)** - за подобрување на перформансите при читање на податоците.

Spring Security

Веб апликациите може да се заштитат во сите од слоевите како:

- **Заштита на мрежен слој** – со предефинирани IP адреси, филтрирање на пакети со firewall и Access Control lists.
- **Заштита на транспортен слој** – заштита на порти, Secure Sockets Layer(SLL) – HTTPS
- **Заштита на ниво на ОС** – доделување минимални привилегии за функционирање на апликацијата, изолација на апликациите (Пр. контејнеризација)
- **Заштита на Апликациски слој** – автентикација(идентитет) и авторизација(пристап)

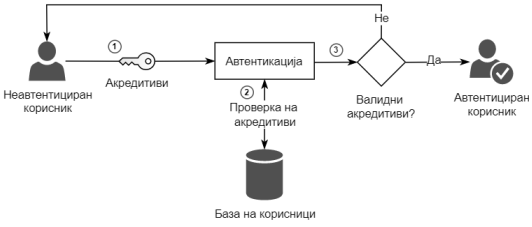
Ние ќе ги искористиме механизмите од **Spring Security** за заштита и безбедност со повеќе внимание кон процесите за автентикација(authentication) и авторизација(authorization) па ќе ги заштитиме следните:

- **Интегритет** – да се осигураме дека информациите при пренос нема да бидат променети од неовластено лице
- **Автентичност** – да ги идентификуваме идентитетот на лицето кој ја пристапува веб апликацијата
- **Доверливост** – да се осигураме дека пораките/податоците се достапни за гледање само за овластени лица
- **Приватност** – да се контролира употребата на личните податоци
- **Достапност** – да се контролира кој може да ја пристапи апликацијата.

Најчести напади на веб апликации се:

- **Вметнување на SQL** – се нарушува **интегритет** и **приватност**, прави промена во базата на податоци неовластено
- **Cross-site scripting (CSS)** – се нарушува **интегритет** и **приватност**, вметнува извршен код како податок и ако се изврши кај друг корисник, може да пристапи до негови податоци и да ги украде или промени.
- **Cross-site Request Forgery (CSRF)** – се нарушува **автентикација** и **доверливост**, напаѓачот користи податоци кои се локално зачувани (колачиња за login…) и со нив извршува акции без знаење на корисникот.

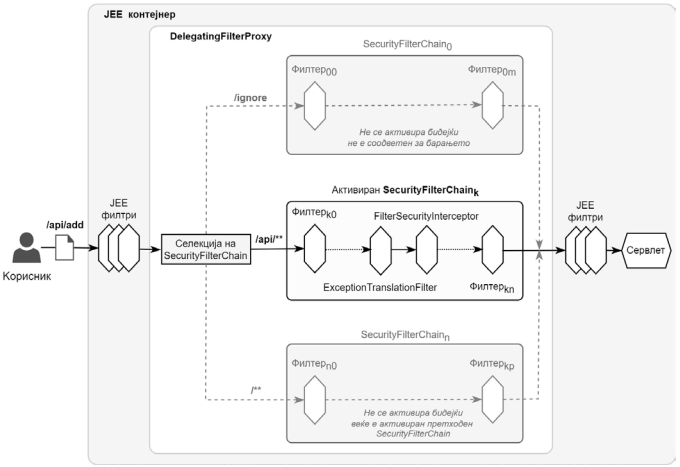
Корисник(User) – корисникот кој е идентификуван и треба да му биде дозволен пристап до апликацијата.
Акредитиви(Credentials) – начинот преку кој корисникот ќе докаже дека тој е оној што вели дека е.
Улога(Role) – групирање на корисниците за да се ограничи кои дозволи корисник ги има ако е во таа група.
Дозволи(Permissions) – Кажуваат кои ресурси се дозволени за одредени улоги или корисници.
Ресурси(Resource) – она што сакаме да го заштитиме



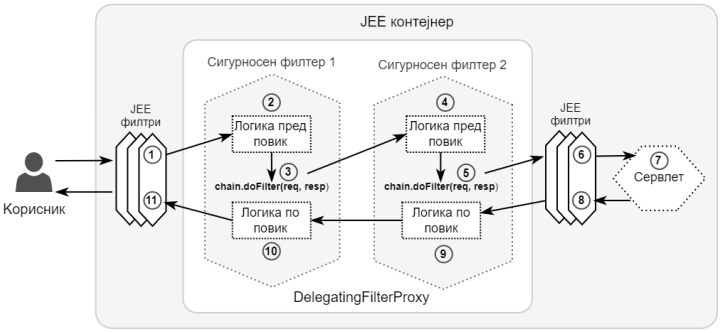
Процеси во безбедност се:

- **Автентикација** - корисникот кажува кој е. Најчесто ова е login process. Корисникот праќа (username и лозинка) или JSON Web Token (JWT) или X509 сертификат. Ако се валидни тогаш се е добро ако не тогаш се враќа статусен код 401 (Unauthorized) и го враќа назад на login страната.
- **Авторизација** - ограничуваме кои корисници имаат пристап до некој ресурс и со проверка во база која е неговата улога и дали е доволна за да може да пристапи до тие страни. Ако ресурсот кој корисник го бара не му е дозволен тогаш се враќа статусен код 403 (Forbidden).
- **Енкрипција** – процес каде сензитивни информации се прават неразбирливи за напаѓачите. Ова е поразлично од енкрипцијата во транспортен слој со SSL (HTTPS). Тука сакаме да ги енкриптираме податоците зачувани во база за да се заштитат од видливост ако се случи data breach.

Заштита на веб апликации со користење на **Spring Security**. Се додава дополнителен слој околу веб апликацијата која ги покрива специфични влезни точки во бизнис логиката со одредени безбедносни правила. Овие правила се конфигурабилни за авторизација на корисниците. Се заштитува од најчестите напади. А конфигурацијата за автентикација е едноставна (Се користат филтри со единствена одговорност за секоја функција).

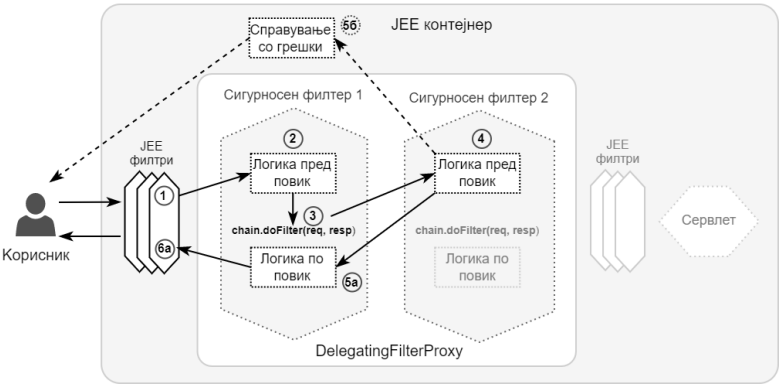


Имплементацијата на заштита преку Spring Security се базира на користење на JEE филтри. Секое барање прво стигнува кај **DelegatingFilterProxy** каде тој селектира и повикува синџир од филтри зависно од патеката на бараниот ресурс. Имплементација на овој синџир на филтри е **SecurityFilterChain** и секој синџир содржи низа од конфигурирани имплементации на **Filter**. Но овие филтри **не се регистрирани** во JEE како стандардни, туку се управувани од Spring Security. Овие филтри се вклучуваат во апликацијата со анотацијата **@EnableWebSecurity**.



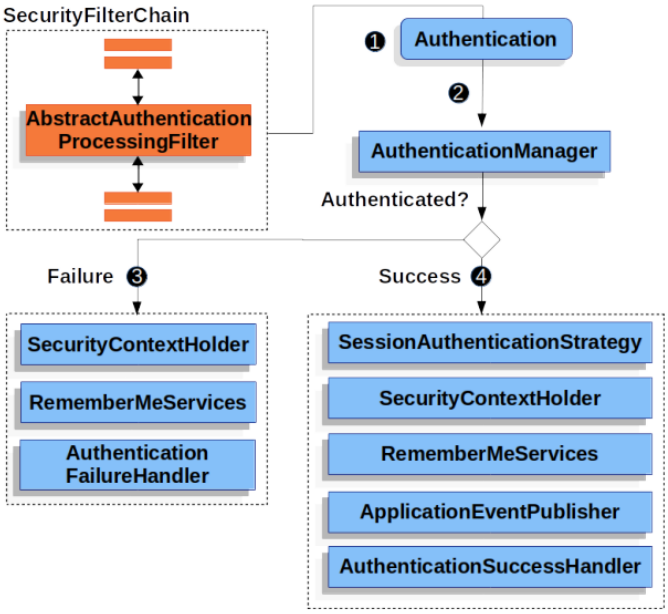
По избирањето на синџирот на филтри, се повикуваат филтрите еден по еден каде во секој филтер има код кој се извршува пред да се препрати барањето на следниот филтер и код кој се извршува после пристигнувањето на одговорот на тоа барање, 'Логика пред повик' и 'Логика по повик'.

Но ако при обработка на еден од филтрите, истиот не дава барањето да продолжи поради некоја грешка или незадоволување на некој критериум. Тогаш филтерот ќе врати грешка и корисникот ќе добие резултат дека има грешка или проблем при обработка на неговото барање и со тоа барањето од тој филтер нема да продолжи по синџирот или пак до сервлетот/контролерот ,туку ќе се врати назад кон корисникот со грешка.

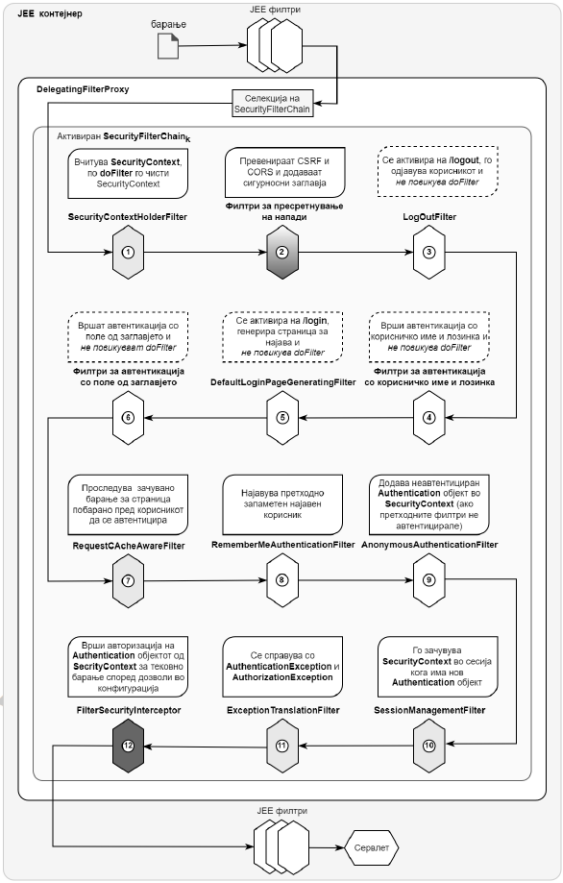


Филтрите кои се дел од **Spring Security** модулот може да се класифицираат во неколку групи:

- **Помошни филтри** – за вчитување и зачувување на податоци потребни во процесот на автентикација и авторизација. Функции на вакви филтри се: пренасочување на неавтентицирани users, справување со грешки
- **Филтри за пресретнување напади** – за превенција на CSRF и CORS(Cross Origin Request sourcing) напади и со додавање на сигурносни заглавја за зголемување на сигурноста.
- **Филтри за автентикација** – за процесот на автентикација и тука може да бидат одговорни повеќе филтри. Во оваа категорија се и филтрите за прикажување на страната за најава и филтрите за одјава.
- **Филтри за авторизација** – за доделување пристап до ресурсите. Претставник на оваа група филтри е **AuthorizationFilter** кој проверува дали на корисникот му е дозволен пристап до ресурсот. Филтрите за автентикација се извршуваат пред овој филтер.

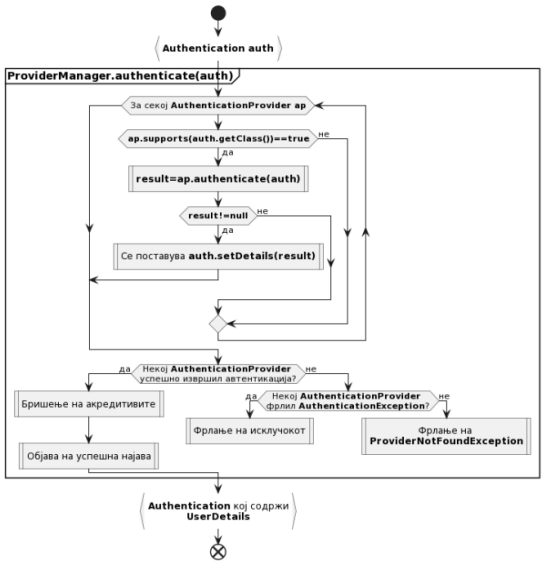


1. **SecurityContextHolderFIlter** – проверува дали има Authentication објект во SecurityContext и истиот го става во SecurityContextHolder (се користи за пристап на SecurityContext) на крај на барањето (по doFilter командата) овој филтер го чисти SecurityContextHolder за нишката да може да се искористи пак ако треба.
2. **CORS and CSRF Filter** – за превенција на вакви напади. Оневозможен е ако го има ова *.cors(AbstractHttpConfigurer::disable)*.
3. **LogOutFilter** – кога барањето е кон /logout или некоја custom патека за logout и филтерот го одлогира корисникот и го чисти SecurityContext и ја инвалидира сесијата.
4. **UsernamePasswordAuthenticationFilter** – кога се пушта POST барање на /login или некоја custom патека за login и креира нов Authentication објект кој се става во SecurityContext и се користи за автентикација. Но ако корисникот внесе погрешни информации тогаш се генерира исклучок *AuthenticationException*.
5. **DefaultLoginPageGeneratingFIlter** – кога се пушта GET барање на /login ама само ако **НЕМА custom login page**.
6. **AuthenticationWithHeaderFieldFilter** – кога барањето има authentication header и апликацијата подржува токени за автентикација. Креира нов Authentication кој се става во SecurityContext и се користи за автентикација.
7. **RequestCacheAwareFilter** – кога корисникот не е автентициран и кога во config а има линијата .defaultSuccessUrl("/home", false) е ваква и вредноста е false. Овој филтер го складира патеката која првично била побарана и потоа го редиректира корисникот ако се најави.



8. **RememberMeAuthenticationFilter** – кога Remember Me функционалноста е активна преку "Remember Me" токен се автентицира. Креира нов Authentication објект кој се става во SecurityContext и се користи за автентикација.
9. **AnonymousAuthenticationFilter** – кога нема Authentication објект во контекстот (т.е. корисникот не е автентициран) и се креира нов anonymous Authentication (AnonymousAuthenticationToken) објект кој се става во SecurityContext.
10. **SessionManagementFilter** – се активира ако претходните филтри 4 или 6 или 8 креираат нов Authentication објект и истиот го складира во сесија. Ако корисник првично се логира со филтер 4 тогаш тоа ќе биде зачувано во сесија и потоа филтерот SecurityContextPersistenceFilter (не е споменат во книга) го зема и го става тоа во SecurityContext.
11. **ExceptionTranslationFilter** – се активира кога има генериран *AuthenticationException* или *AuthorizationException* и потоа се справува со нив и редиректира назад. Овој филтер нема логика пред doFilter командата туку после и тогаш кога некои од филтрите генерираат исклучок
12. **FilterSecurityInterceptor** – се активира кога се барањето е до ограничен(restricted) ресурс. Овој филтер проверува дали корисникот е авторизиран и ако не е тогаш генерира исклучок(*AuthorizationException*).

AuthenticationManager е интерфејс кој дефинира како се врши автентикацијата кај филтрите. Тој има еден метод authenticate кој враќа објект Authentication. Имплементација на овој интерфејс е ProviderManager. **ProviderManager** има листа од конкретни имплементации за ProviderManager и секоја од нив се изминува се додека не се изврши успешна автентикација (со ова се наоѓа со **supports(.Class)** конкретната ситуација за автентикација (anonymous, Remember me, Testing, Dao, Abstract)).



DaoAuthenticationProvider е најчестиот AuthenticationProvider (на пониско ниво на апстракција работи за разлика од AuthenticationManager) кој врши најава со **username** и **password**. Зема UserDetails од UserCache или од UserDetailsService и потоа го валидира (дали е **заклучен** или **истечен**, **валидна лозинка**, **credentials се нови** и не се истечени), па го зачувува во UserCache и креира нов автентициран Authentication објект со UserDetails.

Со **FilterSecurityInterceptor** може да направиме заштита на методите. Се имплементира овој интерцептор преку концептот на Аспект Ориентирано Програмирање (АОР), каде секој метод се енкапсулира.