



Универзитет „Св. Кирил и Методиј“ во Скопје
ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И
КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

Основи на веб програмирање со користење на Spring рамката

Ристе Стојанов, Сашо Граматиков, Милош
Јовановиќ, Димитар Трајанов

Скопје, мај 2023

Акроними

AOP - Aspect Oriented Programming
API - Application Programming Interface
APNG - Animated PNG
AVIF - AV1 Image File Format
CRUD - Create, Read, Update, Delete
CSRF - Cross-Site Request Forgery
CSS - Cascading Style Sheets
DI - Dependency Injection
DIP - Dependency Inversion Principle
DNS - Domain Name System
EE - Enterprise Edition
HTML - Hypertext Markup Language
HTTP - Hypertext Transfer Protocol
HTTPS - Hypertext Transfer Protocol Secure
IANA - Internet Assigned Numbers Authority
IDE - Integrated Development Environment
IP - Internet Protocol
ISP - Interface Segregation Principle
JAR - Java ARchive
JDBC - Java Database Connectivity
JDK - Java Development Kit
JEE - Java Enterprise Edition
JPA - Jakarta Persistence API
JPQL - Java Persistence Query Language
JRE - Java Run-time Environment
JSE - Java Standard Edition
JSON - JavaScript Object Notation
JSP - Java Server Pages
JSTL - JSP Standard Tag Library
JVM - Java Virtual Machine
JWT - JSON Web Token
LDAP - Lightweight Directory Access Protocol
LSP - Liskov Substitution Principle
MIME - Multipurpose Internet Mail Extensions
MVC - Model-View-Controller
OCP - Open-Closed Principle
ORM - Object Relational Mapping

POJO - Plain Old Java Object

RFC - Request For Comments

SOLID - Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle

SQL - Structured Query Language

SRP - Single Responsibility Principle

SSH - Secure (Socket) Shell

SSL - Secure Sockets Layer

TCP - Transport Control Protocol

UI - User Interface

URI - Uniform Resource Identifier

URL - Uniform Resource Locator

VPN - Virtual Private Network

WWW - World Wide Web

XML - Extensible Markup Language

Содржина

Листа на слики	v
Листа на табели	v
1 HTTP	5
1.1 HTTP комуникација	7
1.2 Структура на URL адреса	11
1.3 HTTP пораки	12
1.4 HTTP верзии	14
1.5 HTTP методи	15
1.6 HTTP статусни кодови	17
1.7 Полинња на HTTP заглавје	18
1.8 Колачиња	19
1.9 HTTPS	20
1.10 Примери за HTTP комуникација	24
2 Основи на J2EE	34
2.1 Што е JEE и зошто е потребна?	34
2.2 Сервлет контејнер	36
2.3 Сервлети	40
2.3.1 Имплементација	40
2.3.2 Регистрација на сервлети и пресликување	42
2.3.3 Животен циклус на сервлет	45
2.3.4 Обработка на барања од сервлети	46
2.3.5 Вежба: Имплементација на бизнис логика на сервлет	48
2.3.6 Интеракција помеѓу сервлети	53
2.4 Управување со состојби	61
2.4.1 Параметри и атрибути	61
2.4.2 Работа со параметри и атрибути во апликациски опсег (ServletContext)	62
2.4.3 Работа со податоци во сервлетски опсег (ServletConfig)	68

2.4.4	Работа со параметри и атрибути во опсег на барање	71
2.4.5	Работа со атрибути во сесиски опсег	75
2.4.6	Генерален преглед на параметри, атрибути и опсези на важење	82
2.4.7	Вежба: Работа со параметри и атрибути во опсези на апликација	83
2.5	Филтри	94
2.5.1	Конфигурација на филтри	96
2.5.2	Имплементација на филтри	100
3	Запознавање со Spring рамката	107
3.1	Spring рамка	107
3.2	Дизајн филозофија на Spring	109
3.2.1	SOLID принципи	111
3.3	Вметнување зависности	117
3.4	Инверзија на контролата со користење на Spring рамката	121
3.4.1	Регистрирање на бинови	122
3.4.2	Животен циклус на биновите	125
3.4.3	Автоматско вметнување на зависности	127
3.5	Опции за Конфигурирање во Spring	130
3.5.1	Животен век на биновите	130
3.5.2	Конфигурација на вредности во Spring	131
3.5.3	Надворешни Конфигурации во Spring	132
3.5.4	Конфигурациски профили во Spring	133
3.6	Шаблони за флексибilen и одржлив дизајн на веб апликации	134
3.6.1	MVC шаблон	135
3.6.2	MVC шаблон за веб апликации	136
3.6.3	Слоевита архитектура	137
3.6.4	Слоевит MVC шаблон за веб апликации	139
3.7	Преглед на основните концепти на Spring рамката	140
4	Spring MVC	141
4.1	Запознавање со Spring MVC	141
4.1.1	WebApplicationContext	143
4.2	Spring MVC анотирани контролери	145
4.2.1	Мапирање со HTTP барањата преку анотацијата @RequestMapping	148
4.2.2	Анотирани методи за справување со барања (Handler Methods)	152
4.3	Spring REST	168
4.3.1	Конзумирање на податоци преку REST API	174

4.4	Процесирање на барање кај Spring MVC	176
4.5	Преглед на функционалностите од Spring MVC модулот	182
4.6	Пример контролери (Вежби)	183
5	Работа со релациони бази на податоци	184
5.1	JPA објектно-релационо пресликување	190
5.2	Ентитети	191
5.2.1	Пресликување на табели	193
5.2.2	Пресликување на колони	194
5.2.3	Пресликување на примарни клучеви	195
5.2.4	Пресликување на основни типови на податоци	198
5.2.5	Енумерации	198
5.2.6	Транзитни полинја	200
5.2.7	Вградени и вградливи објекти	201
5.3	Релации	204
5.3.1	Релација „еден-кон-еден“	206
5.3.2	Релација „повеќе-кон-еден“	211
5.3.3	Релација „еден-кон-повеќе“	212
5.3.4	Релација „повеќе-кон-повеќе“	216
5.4	Архитектура на JPA	220
5.5	Користење на EntityManager	223
5.6	Животен циклус на ентитет	225
5.6.1	Креирање на управувани ентитети	226
5.6.2	Вчитување и освежување на ентитет од базата на податоци	227
5.6.3	Ажурирање на ентитет	228
5.6.4	Бришење на ентитет	229
5.6.5	Методи за работа со перзистентен контекст	229
5.6.6	Извршување на кориснички дефинирани прашања	230
5.6.7	Вежба: креирање и користење на репозиториум за работа со ентитети	231
5.7	Работа со ентитети во релација	237
5.7.1	Читање на ентитети во релација	237
5.7.2	Уредување на ентитети во релација	239
5.7.3	Вежба: работа со ентитети во релација	243
5.8	Spring Data JPA	254
5.8.1	Интерфејси за репозиториуми	257
5.8.2	Конвенции за именување на методи	263
5.8.3	Кориснички дефинирани барања	267
5.8.4	Вежба: Работа со Spring Data JPA	269

6 Заштита на веб апликации	279
6.1 Што сè треба да се заштитува?	279
6.2 Напади на веб апликации	282
6.2.1 Вметнување на SQL	282
6.2.2 Cross-site scripting	283
6.2.3 Фалсификување барања меѓу страници (Cross-Site Request Forgery – CSRF)	283
6.3 Основи на безбедност на системи	284
6.3.1 Терминологија за безбедност	284
6.3.2 Процеси во безбедност	285
6.4 Заштита на веб апликации со Spring Security	288
6.4.1 Процес на автентикација и авторизација кај Spring Security	295
6.4.2 Синџир со сигурносни филтри	307
6.4.3 Автентикација со Spring Security	308
6.4.4 Авторизација со Spring Security	320
6.4.5 Преглед на Spring Security функционалности	324
A Полиња на HTTP заглавје	325
B Објекти за HTTP барање и одговор	331
B.1 HttpServletRequest	331
B.2 HttpServletResponse	333
B.3 Правила за повеќекратни пресликувања на сервлети	335
B.4 Дефиниција на параметри на сервлети	335
B.5 Конфигурација на редослед на вчитување на сервлети	337
B.6 Контекстни настани и слушачи на настани	337
B.7 Настани и слушачи на настани за барање	341
B.8 Животен век на сесија	342
B.9 Пристап до параметри за сесија	343
B.10 Настани и слушачи на настани за сесии	343
B Lombok	345

Листа на слики

1-1 Клиент-сервер архитектура на веб сайт	5
1-2 Клиент-сервер архитектура на веб апликација.	6
1-3 Животот на едно HTTP барање.	10
1-4 Анатомија на URL адреса.	12
1-5 Структура на HTTP барање.	13
1-6 Структура на HTTP одговор.	13
1-7 Генерирање на дигитален сертификат.	22
1-8 Валидација на дигитален сертификат.	22
1-9 Воспоставување на SSL врска.	25
1-10 Пример за HTTP комуникација помеѓу клиент и сервер со GET барање	26
1-11 Пример за HTML форма за генерирање на POST барање	30
1-12 Пример за HTTP комуникација помеѓу клиент и сервер со POST барање	31
2-1 Опслужување на барање за динамички веб сайт (апликација)	35
2-2 Животен циклус на еден проект	39
2-3 Дијаграм на животен циклус на сервлет	46
2-4 Процес на опслужување на барање за динамички ресурс од сервлет контејнер	47
2-5 Процес на опслужување на барање за динамички ресурс од сервлет контејнер	48
2-6 Дијаграм на животен циклус на сервлет	54
2-7 Метод forward()	59
2-8 Метод include()	60
2-9 Преглед на карактеристики на атрибути и параметри	62
2-10 Преглед на животен век на ServletContext и неговите параметри	67
2-11 Преглед на животен век на ServletConfig и неговите параметри	70
2-12 Преглед на животен век на ServletConfig и неговите параметри	71
2-13 Преглед на HttpServletRequest и неговите параметри и атрибути	74
2-14 Преглед на HttpSession и неговите атрибути	80

2-15 Преглед на HttpSession и неговите атрибути	82
2-16 Преглед на елементи на сервлет контејнер со параметри и атрибути на различни опсези	84
2-17 Почетна состојба на апликацијата	89
2-18 Состојба на апликацијата откако ќе се повика "/s1?page=1"	90
2-19 Состојба на апликацијата откако ќе се повика "/s2?page=2"	91
2-20 Состојба на апликацијата откако ќе се повика "/s3"	92
2-21 Состојба на апликацијата откако повторно ќе се повика "/s1?page=1"	93
2-22 Состојба на апликацијата откако повторно ќе се повика "/s1?page=1" од различен корисник	93
2-23 Претпроцесирање и постпроцесирање на барање и одговор со користење на филтри	96
2-24 Редослед на извршување на код од филтри при обработка на барање	105
2-25 Комплетен преглед на управувани компоненти, параметри и атрибути	106
3-1 Модули од Spring рамката	109
3-2 Зависности на модулите од Spring рамката	110
3-3 Животен циклус на Spring бинови	126
3-4 MVC шаблон	135
3-5 MVC шаблон за веб апликации	136
3-6 Слоевит MVC шаблон за веб апликации	139
4-1 Хиерархија на контексти кај Spring MVC	144
4-2 Процесирање на барање кај Spring MVC	177
4-3 Припрема на барање кај Spring MVC	178
4-4 Извршување на HandlerExecutionChain	180
4-5 Спраување со исклучоци, исцртување на прегледот и завршување на процесирањето	181
5-1 Апстракција на податоци при работа со бази на податоци	189
5-2 Објектно-релационо пресликување на класа во табела	191
5-3 Табела добиена од пресликување на ентитетот Product	193
5-4 Табела добиена од пресликување на ентитетот Product	202
5-5 Табела добиена од пресликување на ентитетот Company со редефиниција на својства на вградлива класа Address	204
5-6 Табели добиени од пресликување на ентитетите Customer и Address со врска еден-кон-еден	207

5-7	Шематски приказ на дефиниција за двонасочна релација еден-кон-еден	210
5-8	Табели добиени од пресликување на ентитетите <code>Order</code> и <code>Customer</code> со врска повеќе-кон-еден	212
5-9	Табели добиени од пресликување на ентитетите <code>Customer</code> и <code>Order</code> со еднонасочна врска еден-кон-повеќе	214
5-10	Шематски приказ на дефиниција за еднонасочна и двонасочна релација еден-кон-повеќе	216
5-11	Табели добиени од пресликување на ентитетите <code>Product</code> и <code>Category</code> со еднонасочна врска повеќе-кон-повеќе	217
5-12	Шематски приказ на дефиниција за двонасочна релација повеќе-кон-повеќе	219
5-13	Составни компоненти на архитектурата на JPA	220
5-14	Модел на животен циклус на ентитет	225
5-15	Преглед на Spring Data модули	256
5-16	Преглед на репозиториуми	261
5-17	Споредба на методи од <code>JpaRepository</code> и JPA	262
6-1	Процес на автентикација	286
6-2	Процес на авторизација	287
6-3	Преглед на заштита со Spring Security	289
6-4	Повеќе синџири со филтри	290
6-5	Повикување на сигурносните филтри во нормално сценарио	293
6-6	Повикување на сигурносните филтри при исклучок или неиспользовање на безбедносни услови	294
6-7	Преглед на сигурносните филтри од Spring Security модулот	298
6-8	Структура на <code>SecurityContextHolder</code>	299
6-9	Обработка на барања од <code>ExceptionTranslationFilter</code>	301
6-10	Обработка на барања од <code>DefalutLoginPageGeneratingFilter</code> за сценарија без и со конфигурирана предефинирана патека за логирање	303
6-11	Преведување на исклучокот <code>AuthenticationException</code>	305
6-12	Обработка на барања од <code>RememberMeAuthenticationFilter</code>	306
6-13	Структура на <code>ProviderManager</code>	310
6-14	Автентикација со <code>ProviderManager</code>	312
6-15	<code>DaoAuthenticationProvider</code>	316
6-16	Обработка на барање со акредитиви за автентикација преку имплементација на <code>AbstractAuthenticationProcessingFilter</code>	319

Листа на табели

1.1	Опис на HTTP методите. Секое барање покрај метод содржи и ресурс кој се бара од серверот.	16
1.2	Опис на HTTP статусните кодови. Секој одговор од серверот содржи точно еден статусен код.	17
3.1	Модули од Spring рамката	108
5.1	Клучни зборови за дефинирање на пребарувања преку имиња на методи во Spring Data	265
5.2	Клучни зборови за дефинирање на пребарувања преку имиња на методи во Spring Data	266
5.3	Клучни зборови за модифицирање на пребарувања преку имиња на методи во Spring Data	266

Ваш придонес за книгата

Почитувани студенти,

Оваа книга е во работна верзија и е подложна на постојани промени. Со цел да стигнеме до финална и подобрена верзија за издавање, потребни ни се вашите забелешки и мислења. На формата на овој линк <https://forms.office.com/e/ALLgyNxFqz?origin=lprLink> ќе можете да ги внесете грешките кои што ги пронашле, но исто така и проблемите со кои сте соочиле во текот на читањето. Секако, добредојдено ќе биде вашето генерално мислење за секоја глава, без разлика дали е позитивно или негативно. Целта е да се направи учебник наменет за вас, студентите, па затоа вашето мислење и сугестиии во голема мера ќе придонесат кон остварување на таа цел.

Од авторите.

Предговор

Во дигиталниот и поврзан свет, глобалниот пристап до информации и електорнски услуги (е-пошта, е-трговија, е-банкарство, е-здравство, социјални мрежи и сл.) се неизбежен и неопходен дел од нашето секојдневие. Додека Интернетот, како глобална мрежа на поврзани компјутери, ни ја обезбедува инфраструктурата за користење на овие сервиси и пристап до информациите, за нивната имплементација е потребно да се развијат веб апликации кои ќе генерираат динамички содржини прилагодени на нашите барања и преференции. Веб апликациите се хостирали на сервери со кои едноставно остваруваме интеракција преку веб прелистувачите, без разлика дали станува збор за пристап преку персонален компјутер или паметен телефон. Токму поради нивната сестраност и едноставност за употреба, веб апликациите неминовно ги потиснуваат традиционалните десктоп апликации. Тие се причина да се креира сосема нова бизнис околина во која секојдневно се појавуваат нови сервиси за кои сме сведоци дека предизвикуваат значителни промени во секојдневниот начин на функционирање. Оттука, потребата за развивање на нови веб апликации како и за подобрување и одржување на тековните веб апликации е во вртоглав раст.

Иако содржините на динамичките страници од веб апликациите кои се доставуваат до корисниците се базираат на стандардни технологии како што се HTML за дефинирање на елементите на страниците, CSS за дефинирање на нивниот изглед и Javascript за подобрување на однесувањето и интеракцијата со крајниот корисник, постојат различни програсмки јазици и работни рамки кои може да се користат за развивање на веб апликации како што се Java, PHP, Python, C#, Node.js, Ruby итн. Изборот на јазик зависи од многу фактори поврзани со апликацијата, но и од личната преференца на развиваачот на софтвер. За веб апликациите во оваа книга ќе го користиме програмскиот јазик Java, бидејќи е еден од најраспространетите јазици кога станува збор за развој на веб апликации. Дополнително, во рамките на оваа книга ќе биде разработена и Spring рамката, која овозможува брз и флексибilen развој на сигурни веб апликации. Оваа рамка е избрана поради широката распространетост (особено во нашата држава), следењето на најдобрите практики за градење на сигурени и одржливи

веб апликации, како и огромната зедница која помага во надминување на сите потенцијални предизвици и проблеми. Сепак, за да овозможи рапиден развој на веб апликации, Spring рамката абстрагира голем дел од деталите кои се неопходни за да се разбере суштината на развојот на веб апликациите. Поради тоа, во оваа книга се обработуваат и фундаменталните J2EE технологии над кои е изградена Spring MVC библиотеката за развој на веб апликации. На овој начин, читателите на оваа книга ќе се здобијат со вештини за ефикасен развој на веб апликации со Spring рамката, но ќе бидат опремени и со фундаменталните знаења за да можат ја разберат како детално се извршуваат кодот кој го напишале.

Книгата го сублимира долгодишното искуството на авторите во едукативна и апликативна дејност од областа на веб програмирање. Во текот на овој период, авторите ги согледуваат главните потешкотии со кои се сретнуваат студентите при развивањето на веб апликациите и побарувањата на индустриската за подготвени кадри кои ќе можат самостојно да развијат веб апликации. Затоа во континуитет го прилагодуваат материјалот со цел да го доближат до студентите на најдостапен и најразбиралив начин, но истовремено и да одговорат на предизвикот да ги подготват да бидат конкурентни на пазарот на труд. Токму затоа, книгата е замислена како основно и единствено помагало за совладувањето на комплетниот процес на развивање на веб апликации на едноставен и сликовиот начин, елиминирајќи ја потребата читателот да троши дополнително време пребарувајќи низ огромната база на достапни материјали во форма на учебници, блогови или форуми со цел да ги извлече базичните и неопходни концепти на веб апликациите.

Во оваа книга ќе бидат детлно описани концептите на веб апликациите, тргнувајќи од претпоставката дека читателите немаат претходно искуство на темата. Сепак, за да може успешно да се следи содржината, потребни се познавања од објектно ориентираното програмирање со програмскиот јазик Јава, основни познавања маркирачкиот јазик HTML, работа со релациони бази на податоци и работа со интегрирани околини за развивање.

Книгата е наменета за студентите на Факултетот за информатички науки и компјутерско инженерство при Универзитетот „Св. Кирил и Методиј“ во Скопје, запишани на студиските програми на кои се изучува курсот Веб програмирање како задолжителен или изборен предмет. Дополнително, книгата можат да ја користат студентите од останатите факултети од областа на информатички науки ширум Северна Македонија, професионални програмери и ентузијасти кои сакаат да се осposобат да изработуваат веб апликации во рамката Spring.

По совладување на материјалот од книгата, читателите ќе стекнат детално познавање на комуникацијата помеѓу клиент и сервер на апликациско ниво, познавање на рамката Spring и нејзините модули за работа со веб апликации и

ќе бидат во можност самостојно да креираат безбедни веб апликации и сервиси поврзани со релациони бази на податоци.

Цели на оваа книга се:

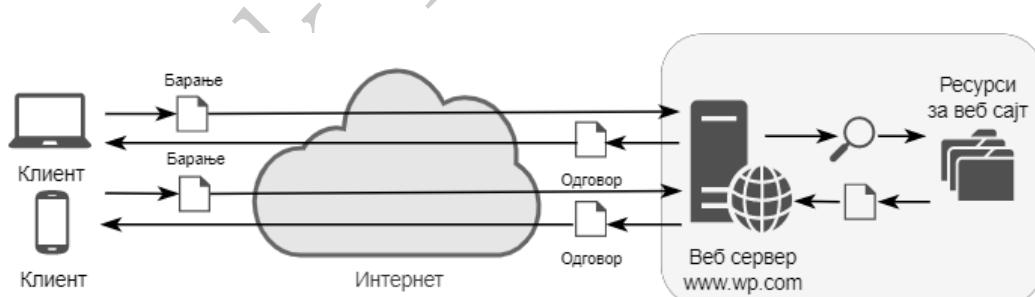
- Запознавање со суштинските концепти за развој на веб апликации и нивно користење при креирање на практична апликација.
- Да ја модернизира Head First – Servlets and JSP
- Да ги селектира и поедностави основните концепти обработени во Pro Spring MVC with Web Flow.
- Да ја ги селектира и да ги постави во поширок контекст концептите дефинирани во Craig Walls - Spring in Action, 5th Edition (2018, Manning Publications).

Глава 1

HTTP

World Wide Web, познат и како WWW, е еден од најраспространетите сервиси на Интернет кој им овозможува на клиентите да пристапат до страниците на веб сайтовите широком светот. WWW го користи моделот клиент-сервер, каде еден сервер опслужува голем број на клиенти. Иако WWW често се поистоветува со Интернет, тој преставува само еден од многуте мрежни сервиси кои го користат Интернетот како мрежна инфраструктура преку која клиентите и серверите остваруваат меѓусебна комуникација.

Под **клиент** (client, англ.) подразбирајме компјутер, лаптоп, паметен телефон, итн. кој поседува **веб прелистувач** како апликација преку која испраќа **барања** за ресурси до **серверот** (server, англ.) и преку која ги прегледува добиените содржини од **одговорот од серверот** (слика 1-1). Примери за веб прелистувачи се Chrome, Safari, Firefox, итн.



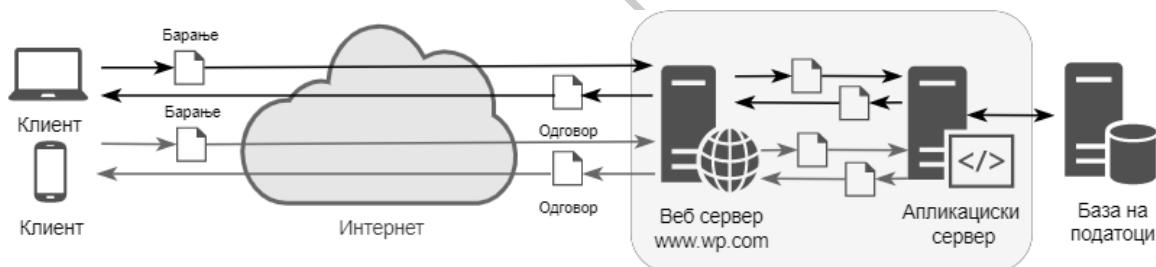
Слика 1-1: Клиент-сервер архитектура на веб сайт.

Веб сайт е колекција од меѓусебно поврзани **веб страници** кои содржат текст, слики, видео, аудио, скрипти и слично. Страниците претставуваат колекција од статички ресурси со визуелни содржини кои корисниците можат да ги читаат и прегледуваат. Секој ресурс е идентификуван преку уникатна адреса наречена **Uniform Resource Locator (URL)**. Почетниот дел од оваа адреса се користи од прелистувачот за да го пронајде серверот на кој се наоѓа ресурсот, а

остатокот од адресата се користи од самиот сервер за да го разликува бараниот ресурс од останатите ресурси кои ги поседува.

Веб сајтовите им се достапни на корисниците преку **веб сервер** кој претставува посебен софтвер инсталiran на компјутер со улога на веб сервер. Веб серверот ги содржи сите ресурси кои ги сочинуваат веб сајтовите, па неговата главна улога е да ги обработува барањата од корисниците за веб страници и да ги опслужи на тој начин што ќе им ги достави сите ресурси кои ги побаруваат. Познати софтвери за веб сервери се Apache, Nginx, Microsoft IIS, итн.

Освен веб сајтови, чии ресурси се статички и оттука секое барање за еден ист ресурс од различни клиенти резултира со иста доставена содржина, постојат и **веб апликации** чии ресури имаат динамички карактер. Во зависност од податоците кои ги испраќа корисникот, веб серверот враќа динамички генерирана содржина која веб прелистувачот му ја прикажува на корисникот на ист начин како и кога станува збор за веб страници. Бидејќи веб серверите можат да опслужуваат само статички ресурси, за да можат да работат со веб апликации тие треба да содржат дополнителен софтвер со улога на апликациски сервер за динамичко генерирање на содржини. Познати софтвери за апликациски сервери се Tomcat, Glassfish, Nginx, Microsoft IIS, итн. Веб апликациите ги генерираат содржините врз основа на податоци сместени во бази на податоци, па затоа, апликацискиот сервер комуницира со сервер на бази на податоци (слика 1-2).



Слика 1-2: Клиент-сервер архитектура на веб апликација.

Без разлика на типот на содржини кои се разменуваат помеѓу клиентот и серверот, или пак, за каков клиентски или серверски софтвер станува збор, за да се оствари успешна и недвосмислена комуникација, потребни се стандардизирани правила кои ќе ги почитуваат двете страни.

Комуникацијата помеѓу клиентите и серверите е дефинирана во **Hypertext Transfer Protocol (HTTP)** протоколот. Овој протокол овозможува пренесување на означен текст (Hypertext) преку компјутерска мрежа. Протоколот дефинира дека комуникацијата помеѓу клиентот и серверот ќе се одвива на следниот начин: (1) клиентот креира и испраќа барање (request, англ.) до серверот, (2) серверот го процесира барањето и (3) го праќа резултатот назад до клиентот во

форма на одговор (response, англ.). Со тоа, велиме дека серверот го *услужува* (serve, serves, англ.) клиентот. Содржината на барањето и одговорот може да варира, но генерално содржи означен текст и хипермедија¹. HTTP протоколот го дефинира начинот на кој се структурираат и разменуваат овие пораки (барањето и одговорот) помеѓу клиентот и серверот, за да можат да бидат недвосмислено разбрани од двата уреди.

Првично, HTTP протоколот најмногу се користел за навигација низ множество од веб страници. Во тој момент, **веб страниците** претставуваат текстуални документи со означен текст, но набрзо нивната содржина се з bogатува со хипермедија, како што се слики, видеа и аудио содржини. Освен содржините кои имаат можност да ги пренасочат корисниците на други ресурси, страниците можат да содржат и форми со полиња за внес на податоци дефинирани во посебни **HTML (Hyper Text Markup Language)** ознаки. Откако корисникот ќе кликне на некој линк, прелистувачот генерира **HTTP барање** кон уникатната локација на ресурсот, а веб серверот го доставува **HTTP одговорот** назад до прелистувачот. Ако одговорот содржи HTML означен текст, прелистувачот ги обработува HTML елементите од одговорот и ги прикажува на корисникот. На пример, ако станува збор за форма со полиња за внес на податоци, тогаш прелистувачот ги испраќа податоците во HTTP порака назначувајќи го ресурсот кој треба да ги обработи. По обработка на пораката, веб серверот враќа нови податоци кои прелистувачот ги прикажува. Со оглед на тоа што една HTML страница претставува означен текст кој може да содржи референцирани хипермедија елементи, откако прелистувачот ќе ја обработи самата страница, за секој хипермедија елемент се генерира посебно барање до серверот за негово преземање и прикажување. Според тоа, без разлика за каков тип на ресурс станува збор, комуникацијата преку HTTP протоколот се базира на **генерирање HTTP барање** до серверот кое резултира со соодветен **HTTP одговор**.

1.1 HTTP комуникација

Протоколот HTTP е претставник на апликацискиот слој од свитата на протоколи TCP/IP врз кои е изграден модерниот Интернет. Овој протокол е имплементиран во веб прелистувачите и веб серверите, криејќи ја целата негова комплексност од крајните корисници.

¹ Термините „означен текст“ и „хипермедија“ се однесуваат на текстуалната и мултимедијалната содржина која е достапна на вебот. Префиксот хипер- во овој контекст се интерпретира како „над“, односно текст и мултимедијална содржина кои се супериорни во однос на отпечатениот текст, графика, аудио и видео содржината достапна надвор од вебот, поради можноста за нивно поврзување преку употребата на хиперлинкови.

За да започне HTTP комуникација, потребно е клиентот да испрати барање за ресурс преку прелистувачот на неговиот компјутерски уред. Тоа најчесто се изведува преку експлицитно внесување на URL адресата на ресурсот во веб прелистувачот или преку кликување на некој хиперлинк од веб страница. Потоа, потребно е веб прелистувачот да го преведе доменското име на серверот, кое се содржи во почетниот дел на URL адресата, во конкретна Интернет протокол адреса (IP address, англ.) на веб серверот. За оваа цел се користи систем за разрешување на доменски имиња (**Domain Name System - DNS**), каде преку специјален DNS протокол се добива IP адреса. Потоа, прелистувачот ја користи оваа адреса за да кон неа ја испрати HTTP пораката (барањето).

Со оглед на тоа што HTTP е протокол на апликациско ниво кој знае да обработува само HTTP барања и одговори, но не и да ги пренесе низ недоверливиот Интернет, како транспортен протокол за доверлива размена на податоци помеѓу клиентот и серверот се користи протоколот за контрола на преносот (**Transmission Control Protocol - TCP**). За да го испрати HTTP барањето, клиентот, односно неговиот веб прелистувач, најпрвин отвора TCP конекција со серверот. За да се отвори конекција, клиентот мора да ја специфицира IP адресата на серверот (која веќе ја добил од DNS сервисот) и дестинациската порта на која серверот слуша и очекува барања за конекции. Предефинирана порта за протоколот HTTP е портата 80. Ако портата е различна од предефинираната, тогаш таа мора да се наведе во самата URL адреса.

Серверот ја прифаќа конекцијата, по што веб прелистувачот го испраќа барањето и преку истата конекција го добива одговорот назад. TCP го користи IP протоколот за да ги испрати TCP сегментите низ IP мрежата. TCP сегментите во своето тело ги содржат HTTP пораките. Ако една HTTP порака е преголема за да се смести во еден сегмент, тогаш таа се дели на помали парчиња кои се разменуваат во посебни TCP сегменти. Поради недоверливоста на IP протоколот (не гарантира дека IP пакетите ќе бидат доставени до дестинацијата), TCP протоколот се грижи евентуално изгубените сегменти да бидат препратени, па со тоа му гарантира на HTTP протоколот дека сите пораки ќе бидат разменети во целост, истовремено контролирајќи ја брзината на трансфер да одговара на можностите на мрежата, клиентот и серверот.

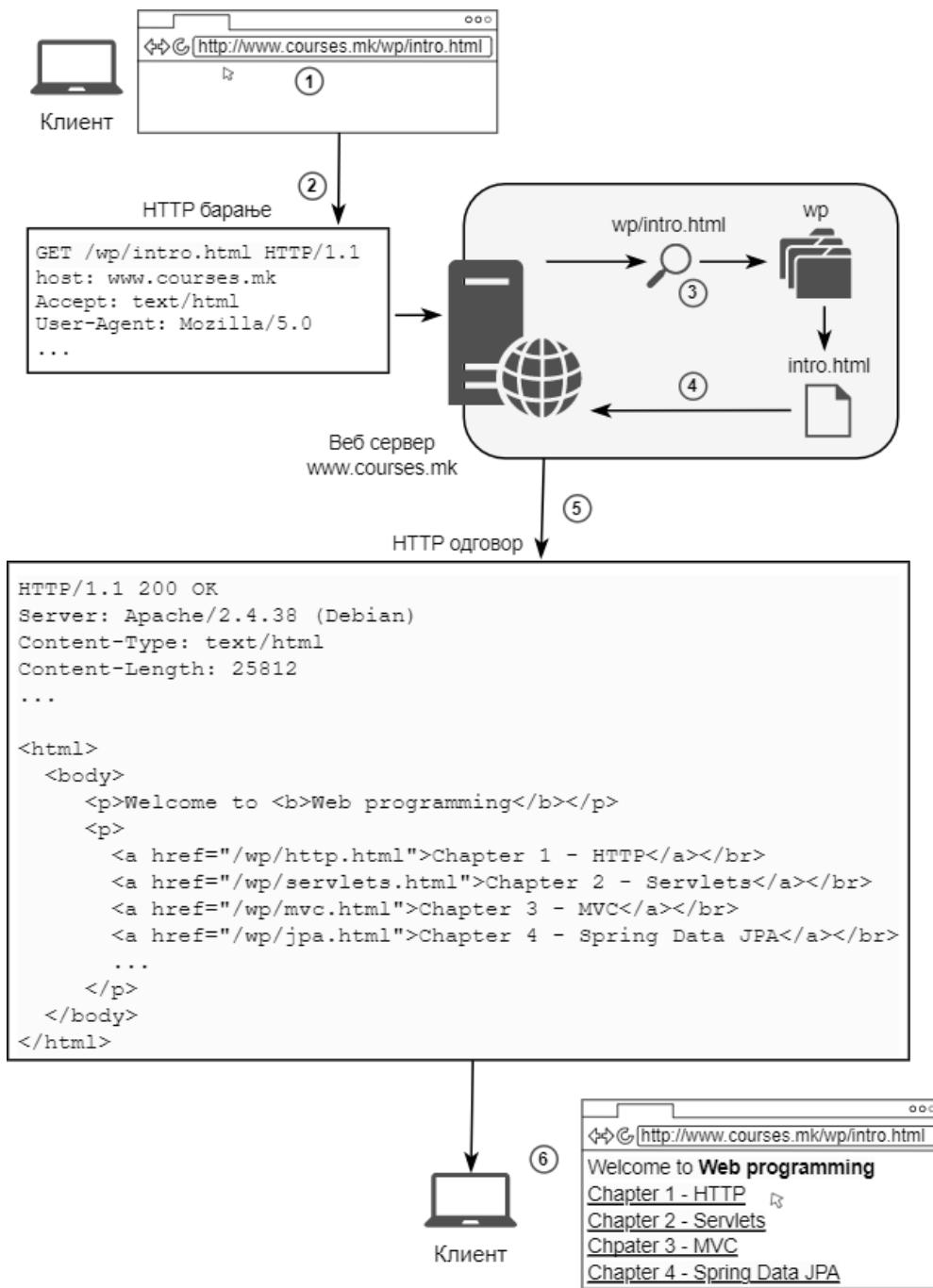
TCP сегментот кој го содржи HTTP барањето се предава на IP слојот, кој по потреба го дели и испраќа во форма на пакети. Секој пакет патува до серверот преку компјутерската мрежа така што попатните рутери го насочуваат до серверот врз основа на неговата IP адреса. Пакетот пристигнува кај серверот, тој ја чита IP адресата, препознава дека пакетот е наменет за него, го отпакува и го препраќа на TCP слојот. Во овој слој, врз основа на заглавјата на TCP сегментот, се одлучува која апликација ќе ја процесира пораката. Бидејќи станува збор за

HTTP барање (дестинациска порта 80 во заглавјето на TCP), тоа се препраќа до веб серверот кој комплетно ја разбира содржината на HTTP барањето, го обработува и одредува која содржина ќе ја врати назад до клиентот преку HTTP одговор.

При оваа комуникација, клиентот преку веб адресата точно прецизира на кој сервер се обраќа и кој ресурс точно го бара. Но, покрај адресата на ресурсот, клиентот го посочува и **типот на барањето** во зависност од тоа дали бара само ресурс или преку барањето испраќа и **кориснички податоци во телото**, како и дополнителни информации преку **заглавјата на барањето** кои му се потребни на серверот за негова правилна обработка. Некогаш може да се случи бараниот ресурс да не е достапен на серверот. Во овој случај, серверот мора да има начин да го извести клиентот дека бараниот ресурс не постои или можеби се наоѓа на друга локација. За оваа цел се користат **статусни кодови** кои на клиентот му помагаат да дознае што точно направил серверот со неговото барање. Покрај статусните кодови, во **заглавјето на одговорот** се испраќаат дополнителни информации кои му се потребни на прелистувачот за да ги обработи испратените податоци во **телото на одговорот**. Подетално за содржината на пораките и статусните кодови ќе зборуваме во следните поглавја.

На слика 1-3 е даден пример за комуникација помеѓу клиент и сервер, каде во фокусот е само размената на пораки преку HTTP без да се навлегува во детали за протоколите на подолните нивоа. Во примерот, клиентот пристапува до страница на еден статички веб сајт, односно испраќа HTTP барање за датотеката `intro.html`, која е сместена во директориум со патека `wp` на серверот со име `www.courses.mk`. Текот на комуникацијата е следен:

1. Клиентот ја внесува URL адресата `http://www.courses.mk/wp/intro.html` на веб страницата во неговиот веб прелистувач.
2. Веб прелистувачот го извлекува доменското име на серверот (`www.courses.mk`) од URL адресата и откако ќе го преслика името во соодветна IP адреса, воспоставува TCP конекција со него, на порта 80. Потоа, креира HTTP барање и го испраќа преку воспоставената конекција. Барањето е од типот GET, што означува дека клиентот само го бара ресурсот без да праќа дополнителни податоци во неговото тело. Адресата на ресурсот во рамките на серверот е останатиот дел од URL адресата после доменското име, односно `/wp/intro.html`.
3. Серверот го прима барањето и ја користи адресата на ресурсот за да го лоцира во неговиот датотечен систем. Конкретно, ја бара датотеката со име `intro.html` зачувана во именикот за сајтот `wp`.
4. Серверот го лоцира ресурсот и креира HTTP одговор во чие тело ја сместу-



Слика 1-3: Животот на едно HTTP барање.

ва содржината на датотеката `intro.html`. Датотеката содржи html елементи за прикажување на обичен текст и на листа од ставки кои претставуваат хиперлинкови кои водат до други ресурси од сајтот. Серверот го испраќа и статусниот код 200 со опис OK за да му назначи на прелистувачот дека успешно го пронашол ресурсот кој го бара. Дополнително, серверот назна-

чува дека содржината на телото е html код со вкупна големина од 25.812 байти.

5. Серверот го испраќа HTTP одговорот до клиентот преку воспоставената конекција.
6. Клиентот го прима HTTP одговорот. Неговиот прелистувач најпрвин го чита статусот кој означува дека успешно е добиен бараниот ресурс, а потоа ја презема целата содржина од неговото тело. Врз основа на заглавјето Content-Type заклучува дека таа содржина е html код кој го рендерира и го прикажува во формат разбиралив за корисникот.

1.2 Структура на URL адреса

URL адресата претставува уникатен идентификатор на ресурси до кои може да се пристапи преку Интернет мрежата. Освен за адресирање на датотеки (.html, .css, .js), URL се користи и за да се пристапи до разни Интернет сервиси (пр. REST сервис). URL е еден од клучните концепти на WWW и не се однесува исклучиво на протоколот HTTP, туку може да се користи и за пристап до ресурси на Интернет мрежата преку други протоколи на апликациско ниво како што се HTTPS, FTP и FTPS.

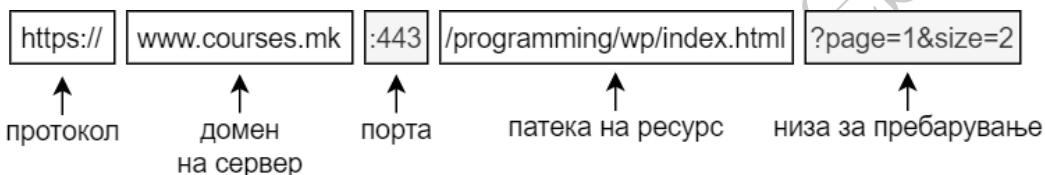
Секоја URL адреса претставува низа од ASCII знаци која не смее да содржи празни места. Сепак, често е потребно низата да содржи знаци кои не припаѓаат на множеството ASCII, или пак содржи празни места. Во тој случај, секој таков знак се конвертира во URL-кодирана низа која се состои од знакот % проследен со два хексадецимални знаци кои го означуваат кодот за знакот. На пример, празното место се заменува со URL-кодираната низа %20 (или со знакот +).

URL адресата се состои од следните елементи (слика 1-4):

1. **Протокол** - го означува протоколот преку кој ќе се преземе ресурсот. Кога станува збор за пристап до веб страници или веб апликации, се користат протоколите HTTP или HTTPS. После името на протоколот се додава низата ://
2. **Домен на сервер** - име на серверот кој го содржи ресурсот. Се користи за наоѓање на IP адресата преку DNS протоколот.
3. **Порта** - ја означува дестинациската порта на која серверот слуша за конекции. Се користи од страна на прелистувачот за да воспостави TCP конекција со серверот. Најчесто се изоставува од URL адресата и во тој случај прелистувачот ја користи предефинираната порта: 80 за HTTP, односно 443 за HTTPS.

4. **Патека на ресурс** - ја означува патеката на ресурсот во рамките на серверот. Оваа патека е релативна во однос на именикот кој е означен како локација за документите на веб серверот.
5. **Низа за пребарување** - Опционална низа која започнува со знакот ? после која следат парови клуч=вредност одвоени со знакот &. Секој пар клуч=вредност претставува кориснички податок кој се испраќа до серверот.

На слика 1-4 е даден пример за URL адреса во која како протокол за преземање на ресурсот се користи HTTPS. Ресурсот се наоѓа на сервер со доменско име `www.courses.mk` кој слуша на порта 443. Во рамките на серверот, ресурсот се наоѓа на патеката `/programming/wp/index.html`. Дополнително, во низата за пребарување прелистувачот го испраќа параметарот `page` со вредност 1 и `size` со вредност 2.



Слика 1-4: Анатомија на URL адреса.

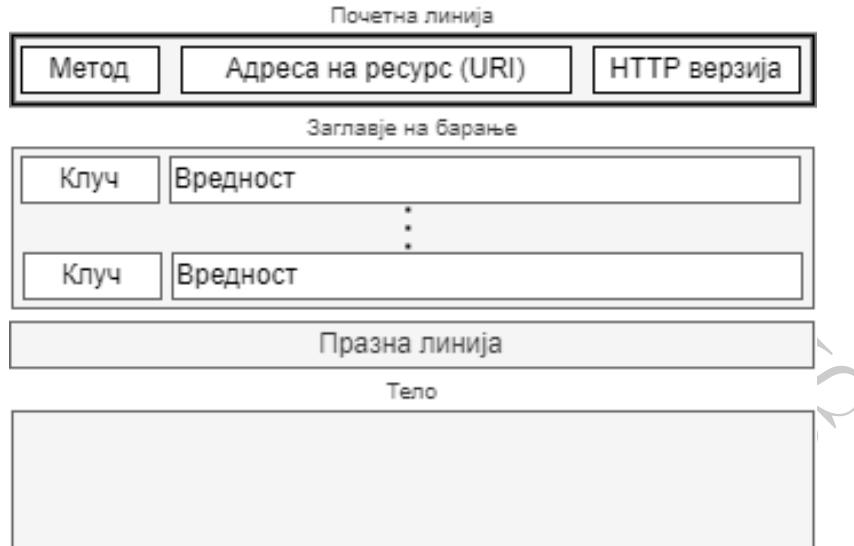
1.3 HTTP пораки

HTTP пораките имаат точно дефинирана структура и можат да бидат дефинирани како HTTP барање доколку потекнуваат од клиентот, или како HTTP одговор доколку потекнуваат од серверот. Структурата на барањето е дадена на слика 1-5. Првата линија е задолжителен дел на барањето и се состои од:

- **Метод** - го дефинира типот на барањето со што му дава информации на серверот како да ги третира податоците кои се испратени во остатокот од пораката и како да ја обработи самата пораката;
- **Адреса на ресурсот** - релативна адреса на ресурсот која се состои од низа на знаци после доменското име на целата адреса;
- **HTTP верзија** - верзијата на HTTP протоколот која ја поддржува прелистувачот.

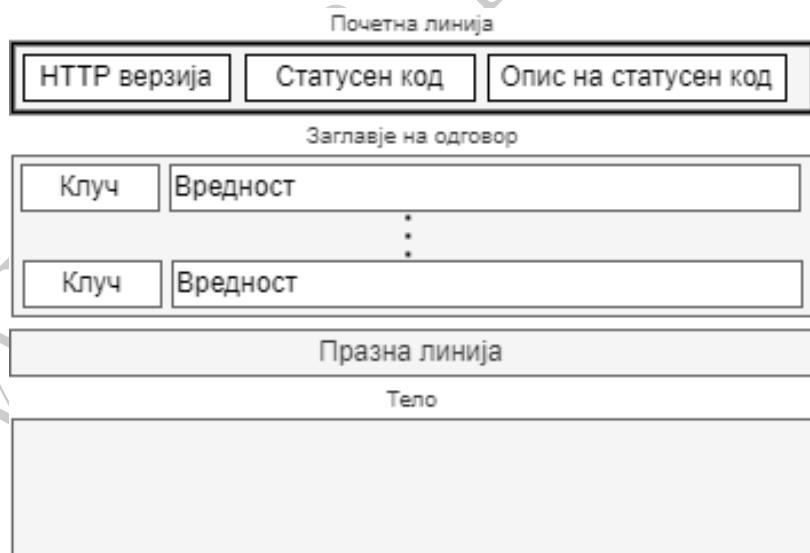
Заглавјето на барањето (header fields, англ.) се состои од клуч-вредност парови кои го допрецизираат барањето или содржат информации за клиентот, содржина која треба да се испроцесира од серверот, метаподатоци за содржината која се испраќа, итн. По заглавјето следува празна линија и на крајот е телото

на барањето. Во зависност од типот на барањето, телото може да биде празно, односно да не постои.



Слика 1-5: Структура на HTTP барање.

Серверот одговара на клиентското барање преку испраќање на еден или повеќе одговори, при што секој од нив ја има структурата прикажана на слика 1-6.



Слика 1-6: Структура на HTTP одговор.

Почетната линија на одговорот се разликува во структурата од почетната линија на барањето. Таа се состои од:

- **HTTP верзија** - верзијата на HTTP протоколот која ја поддржува серверот. Серверот ја зема предвид верзијата која ја испратил клиентот и доколу поддржува повеќе верзии, ја испраќа највисоката верзија која ја поддржува клиентот. Ова поле е потребно за да клиентот и серверот го користат истиот јазик на комуникација;
- **Статусен код** - го означува статусот на одговорот. Се користи од страна на прелистувачот, за да знае на кој начин да го третира одговорот;
- **Опис на статусен код** - текстуален опис на статусот наменет за луѓето, кој носи дополнителни информации за статусот.

1.4 HTTP верзии

HTTP е основен протокол за World Wide Web (WWW) уште од неговото појавување во 1989 година. Оттогаш претрпнува неколку значајни измени кои придонесуваат за подобрување на неговите перформанси и функционалности, во склад со растот на сајтовите, на мрежните и на веб технологиите. Постојат неколку различни верзии: HTTP/0.9, HTTP/1.0, HTTP/1.1 и HTTP/2.0. HTTP/1.1 е првата стандардизирана и стабилна верзија која има најмасовна употреба. Новитетите кои ги воведува оваа верзија се:

- можност за испраќање на повеќе барања преку иста TCP конекција, без таа да се затвора;
- вклучување на комплетното множество на HTTP методи;
- преговарање за типот на содржини;
- компресија на заглавја, итн.

Сепак, со растот на обемот на веб страниците и мултимедиските содржини кои се дел од нив, оваа верзија се соочува со проблемот на закочување од страна на првото барање во редицата (Head of Line Blocking, англ.), односно доколку на серверот му е потребно долго време да го опслужи тековното барање во редот на пристигнати барања, останатите барања не можат да бидат опслужени, што може да резултира со долго време на чекање за прикажување на комплетната веб страна кај клиентот.

Овие проблеми се надминуваат со верзијата HTTP/2.0, која се појавува во 2015 година. Оваа верзија овозможува:

- мултиплексирање на повеќе барања преку една TCP конекција;
- приоретизирање на барања, опслужувајќи ги оние кои се со поголема важност за првично прикажување на елементи на веб страницата, со цел постигнување подобро корисничко искуство;

- понапредна компресија на заглавја; и
- можност серверот самостојно да иницира комуникација со клиентот (push notifications, англ.).

HTTP/2.0 ја поддржуваат познатите веб прелистувачи и веб сервери, па затоа денес, најголемиот дел од веб сајтовите ја користат токму оваа верзија за достава на содржини.

Следната верзија на протоколот е HTTP/3.0, но таа сè уште е во изработка и не е широко прифатена. Се разликува по тоа што наместо TCP, како транспортен протокол го користи експерименталниот протокол QUIC², имплементиран над UDP.

1.5 HTTP методи

HTTP методите се користат за да ја означат намерата на барањето кое даден клиент го испраќа до серверот, како и да прецизираат што всушност очекува клиентот како успешен одговор од тоа барање. Главните методи кои се користат во рамки на HTTP протоколот се **GET** и **POST**. Останатите методи дефинирани во протоколот се **HEAD**, **PUT**, **DELETE**, **CONNECT**, **OPTIONS** и **TRACE**. Преглед на HTTP методите е даден на Табела 1.1.

HTTP GET методот се користи од страна на клиентот за да биде побарана тековната репрезентација на ресурсот на кој се однесува барањето. Доколку, на пример, клиентот користи GET метод за своето барање и барањето се однесува за ресурсот <https://finki.ukim.mk>, тогаш серверот е должен како одговор на барањето да ја врати тековната содржина на овој веб ресурс. Доколку по извесно време се промени содржината на ресурсот (се ажурира веб страната, се објават нови информации, итн.), ново барање со истиот метод кон овој ресурс ќе резултира со нов, различен одговор од страна на серверот, затоа што согласно стандардот, одговорите на барањата кои го користат GET методот секогаш се однесуваат за тековната репрезентација на ресурсот. Важно е да се напомене и дека GET е еден од двата задолжителни HTTP методи (заедно со HEAD), кои согласно стандардот, мора да бидат поддржани од секој HTTP сервер.

Главната цел на методот GET е да го наведе ресурсот (датотека или сервис) преку неговата URL адреса во почетната линија на барањето, а како одговор од серверот да го добие бараниот ресурс. Сепак, постои можност да се пренесат и кориснички дефинирани податоци до серверот, но нивната количината е ограничена, бидејќи единствениот начин да се пренесат е да бидат вклучени како дел од URI адресата на ресурсот. Примери за пренос на податоци во оваа форма

²QUIC <https://datatracker.ietf.org/doc/html/rfc8999>

Метод	Опис
GET	Се бара тековната репрезентација на ресурсот. Во овие барања не се испраќа содржина во телото.
HEAD	Исто како GET, но без пренос на содржината во телото на одговорот кој се враќа т.е. одговорот нема тело.
POST	Се бара обработка на содржината испратена во телото на барањето.
PUT	Се бара да се замени тековната репрезентација на ресурсот со содржината испратена во телото на барањето.
DELETE	Се бара бришење на сите тековни репрезентации на ресурсот.
CONNECT	Се бара креирање на мрежен тунел до серверот означен со идентификаторот на ресурсот.
OPTIONS	Се бара опис на комуникациските можности на ресурсот.
TRACE	Се изведува тест со праќање на пораките до ресурсот и нивно враќање назад до испраќачот, со цел да се одреди патеката по која патуваат податоците.

Табела 1.1: Опис на HTTP методите. Секое барање покрај метод содржи и ресурс кој се бара од серверот.

се: пренос на бројот на страница која ја избрал корисникот од листа на продукти поделена во страници со фиксна големина, текст од пребарување за кој серверот треба да даде резултат, итн. Потребно е да се внимава да не се пренесуваат сензитивни податоци (лозинки, броеви на платежни картички, итн.) преку GET методот, бидејќи дури и да се користи заптитена комуникација (HTTPS), веб серверите водат дненик на обработени барања во кои ја запишуваат првата линија од барањето, па на тој начин веб администраторот има увид во сите податоци кои се пренеле преку URL адресата. Овој метод најчесто се користи при внесување на URL адреса во прелистувачот и при клик на линкови.

HTTP POST методот се користи од страна на клиентот за да биде испратена содржина до серверот, за која се бара да биде обработена од страна на самиот сервер. Притоа, обработката зависи од ресурсот кон кој е испратено ова барање. На пример, некои од најчестите употреби на POST барањата се:

- испраќање блок на податоци (пр. податоци внесени во HTML форма) до процес кој треба да ги обработи,
- испраќање текстуална содржина, коментар, слика или видео содржина на социјална мрежа или блог, со цел нејзино објавување,
- додавање податоци на веќе постоечка репрезентација на ресурс, итн.

1.6 HTTP статусни кодови

Статусните кодови претставуваат троцифрени цели броеви кои го објаснуваат резултатот од барањето и значењето на одговорот, односно означуваат дали барањето било успешно и што претставува одговорот (доколку воопшто е испратен). Валидни статусни кодови се кодовите од 100 – 599.

Првата цифра од статусниот код ја означува класата на одговорот. Втората и третата цифра се користат за да се прецизира резултатот од барањето. Првата цифра може да има една од пет вредности, дефинирани како посебни класи и прикажани во Табела 1.2.

Код	Категорија	Опис
1xx	Информативен статус	Барањето е применено, процесот продолжува.
2xx	Успешно барање	Барањето е успешно применено, разбрано е и е обработено.
3xx	Пренасочување	Потребно е да се преземат дополнителни активности за да се комплетира барањето.
4xx	Клиентска грешка	Барањето содржи погрешна синтакса или не може да биде исполнето.
5xx	Серверска грешка	Серверот не успеа да го обработи валидното барање.

Табела 1.2: Опис на HTTP статусните кодови. Секој одговор од серверот содржи точно еден статусен код.

Едно барање од клиентот најчесто добива еден одговор од серверот кој има точно еден статусен код. Но, согласно стандардот, едно барање од клиентот може да добие и повеќе привремени одговори со статусни кодови од категоријата „информативно“ (1xx), после кои ќе следи точно еден финален одговор со статусен код од една од останатите категории.

Во продолжение ќе разгледаме некои од најчестите статусни кодови со кои се среќаваме при користење на HTTP протоколот, но и при програмирање на веб апликации:

- Статусен код **200** со статусен опис **OK** е најчестиот стандарден одговор кој се добива од веб серверите. Означува дека барањето за ресурсот е успешно и дека бараниот ресурс е во телото на одговорот.
- Статусен код **301** со статусен опис **Moved Permanently** се добива кога корисникот бара ресурс од серверот кој е перманентно преместен на друга

локација. На пример, ако го смениме доменот на веб сајтот, со тоа ја менуваме и локацијата на сајтот. За да може корисниците кои се упатуваат до старата URL адреса да пристигнат до ресурсот на новата URL адреса, веб администраторот на сајтот со старото име поставува правило со кое сите барања до него да бидат пренасочени кон новиот веб сајт, па затоа секое барање со стара адреса резултира со враќање на статусен код 301 и информација за тоа која е новата адреса на која треба да се обрати прелистувачот. По добивање на одговор со ваков статусен код, прелистувачот генерира ново барање кон новата локација.

- Статусен код **400** со статусен опис **Bad Request** означува дека клиентот испраќа податоци до серверот во погрешен формат кој е неприфатлив за серверот. На пример, ако корисникот испраќа текстуална вредност за сервис кој очекува целобройна вредност, сервисот не може да го обработи барањето и го одбива како несоодветно.
- Статусен код **403** со статусен опис **Forbidden** се добива кога корисникот се обидува да пристапи до ресурс за кој нема дозвола. На пример, ако неавтентициран корисник на веб сајт се обиде да пристапи до ресурси за администрација на сајтот кои се достапни само за администраторот, серверот одговара со забрана за пристап поради немање дозвола.
- Статусен код **404** со статусен опис **Not Found** се враќа доколку ресурсот што го бараме не постои на серверот. На пример, ако се згреши името на ресурсот и ако ресурс со такво име не постои на серверот, тогаш тој ни враќа одговор со статус 404.
- Статусен код **500** со статусен опис **Internal Server Error** се враќа кога настанува грешка на серверска страна. Пример за порака со ваков статус е барање за ресурс придржано со податоци кои се во валиден формат, но поради лошо дизајниран софтвер, обработката на барањето предизвикало грешка со која веб апликацијата не успеала да се справи. Одговори со ваков статусен код значат пропуст во апликацијата, па затоа не е пожелно да е појави во продукциска верзија на веб апликација. Причините за грешките во ваквите случаи треба да се бараат во логовите на веб апликацијата, кои во одредени случаи може и да бидат вклучени во телото на одговорот.

1.7 Полиња на HTTP заглавје

Полињата на заглавјето на HTTP барањето и одговорот носат дополнителни податоци со информации за нив и им служат на серверот и клиентот при обработка на пристигнатата порака. Важноста на овие полиња од аспект на веб програмирање е во тоа што, понекогаш, на серверска страна е потребно експлицитно да

се прочитаат нивните вредности за да се знае како да се обработат податоците во барањето, или пак да се постави нивната вредност, со цел да му се назначи на прелистувачот како да ги обработува податоците кои пристигнуваат. Листата на заглавјата е прилично долга, но оние кои се од поголема важност за целите на книгата се описаны во додаток [A](#).

1.8 Колачиња

HTTP по природа е безсостојбен протокол, што значи дека откако серверот ќе добие барање од клиент и го опслужи, ги ослободува сите ресурси кои биле резервирали за негова обработка. Серверот не води никаква евиденција за претходната комуникација со клиентите, па поради тоа кога корисникот повторно ќе генерира барање до истиот веб сайт, серверот го третира барањето независно од претходните, како да е испратено од нов клиент. Ваквото однесување не е практично при секојдневно користење на веб апликациите, бидејќи не овозможува континуитет. На пример, откако ќе се најавиме на веб апликација за е-продавница и внесеме неколку продукти во потрошувачката кошничка, потребно е серверот да ни ја испраќа содржината на кошничката во секое наредно барање или пак при следната најава. Поради тоа што веб серверот не чува информации за клиентите, кај HTTP се користат **колачиња** (cookies, англ.) кои преставуваат серверски податоци кои се испраќаат преку одговорот и се зачувуваат во перманентната меморија на прелистувачот на клиентска страна.

Серверот ги испраќа колачињата како клуч-вредност парови при првата посета на корисникот на веб страницата преку полето **Set-Cookie** од заглавјето на одговорот. За секое колаче серверот поставува и животен век, кој означува колку време прелистувачот ќе го чува колачето пред да го избрише. Ако клиентот поддржува колачиња, тогаш неговиот прелистувач ги испраќа зачуваните колачиња од тој веб сайт во секое барање кое го упатува до него, преку полето **Cookies** во заглавјето. Откако серверот ќе го прими барањето, може да го идентификува клиентот кој го испратил преку неговото колаче. Колачето најчесто го користат веб апликациите по најава на корисниците, за да во него испратат автентикациски токен кој се користи во секое наредно барање, со цел да се одржи активна сесијата на корисникот. Друга типична примена на колачињата е за персонализација на содржинити кои се прикажуваат на корисниците, или пак за водење евиденција на нивното однесување при користење на веб сајтовите (на пример: посетени страници, време на задржување на секоја страна, итн.).

1.9 HTTPS

Протоколот HTTP сам по себе не нуди заштита на пораките кои се разменуваат помеѓу клиентот и серверот. Податоците испратени преку барањето или одговорот можат да бидат пресретнати од потенцијален напаѓач и искористени во злонамерни цели. На пример, ако се најавуваме на веб апликација, корисничкото име и лозинка се пренесуваат преку телото на POST барање кое се испраќа преку TCP, потоа преку IP и на крајот протоколот на податочно ниво и на физичкиот слој, кој ги испраќа битовите до следниот крај на врската. Најчесто IP пакетите патуваат низ рутерите на Интернет неекриптирани, па било кој што ќе се најде на патеката помеѓу клиентот и серверот може да ја прочита нивната содржина, вклучувајќи го телото кое носи податоци од погорните слоеви. Поради тоа, се користи меѓу-слојот Secure Socket Layer (SSL) кој се наоѓа помеѓу HTTP и TCP, чија главна улога е да ја заштити содржината на пораките од HTTP пред да бидат предадени на TCP. HTTPS не е посебен протокол, туку претставува HTTP комуникација преку SSL.

HTTPS (HTTP Secure) е безбедна варијанта на HTTP протоколот која гарантира доверливост, интегритет и автентикација на комуникацијата, односно гарантира дека при комуникацијата помеѓу клиентот и серверот, пресретнатите пораки нема да можат да бидат прочитани или променети и дека серверот го има навистина овој идентитет кој што ни го прикажува (не можеме да знаеме кој се наоѓа од другата страна на комуникацијата).

Секој веб сервер го имплементира протоколот SSL. За да се конфигурира безбедна комуникација, администраторот на веб сајтот најпрво треба да обезбеди безбедносен **дигитален сертификат** кој го добива од посебно тело наречено **сертификациски авторитет (CA)** (certificate authority, англ.).

На слика 1-7 е прикажан комплетниот процес на генерирање на дигитален сертификат за еден корисник.

1. Барателот на дигитален сертификат најпрвин креира пар од **јавен клуч** кој ќе биде јавно достапен и **приватен клуч** кој ќе биде зачуван на безбедна локација на серверот, кој ќе го користи за асиметрична енкрипција. Најчесто користена алатка за генерирање ваков пар на клучеви е `ssh-keygen`³. Оваа алатка генерира приватен клуч во текстуална датотека без екstenзија и јавен клуч во текстуална датотека со екstenзија `.pub`. За потсетување, кај асиметричната енкрипција, пораката криптирана со јавен клуч може да се декриптира само со помош на приватниот клуч, но не и со истиот клуч со кој е криптирана, односно јавниот клуч. Важи и обратното: порака

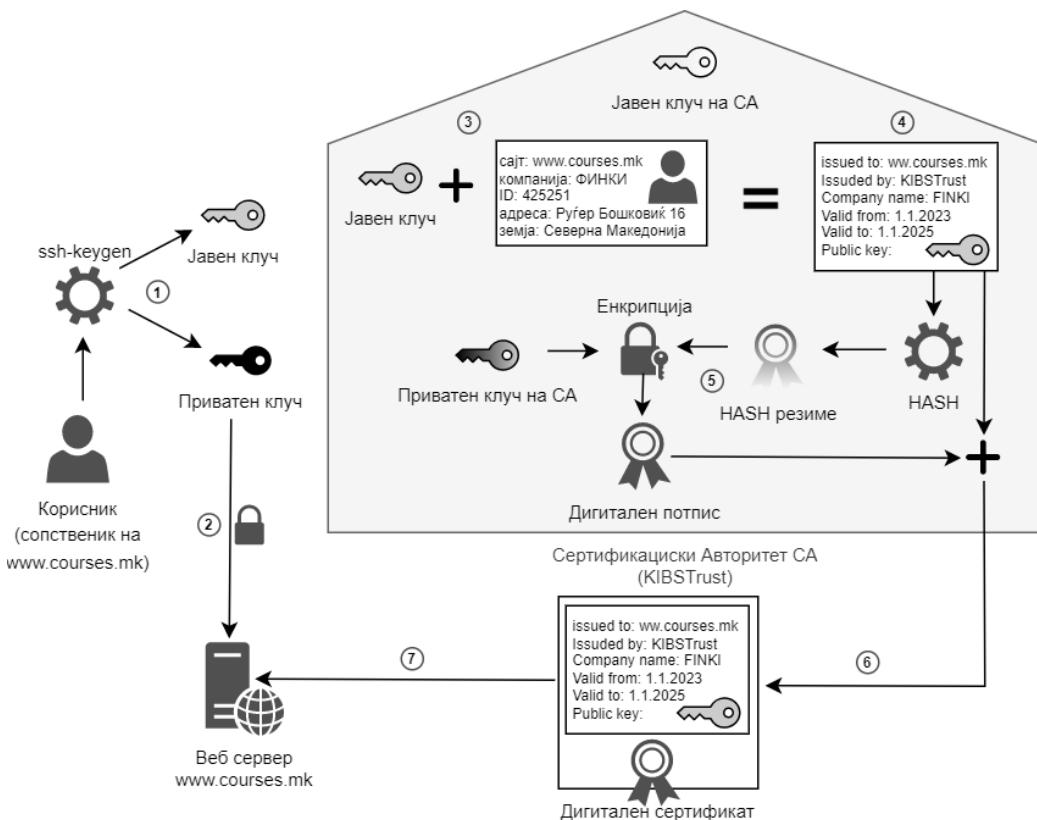
³ssh-keygen: <https://www.ssh.com/academy/ssh/keygen>

криптирана со приватен клуч може да се декриптира само со јавен клуч. Овој тип на енкрипција бара извесно процесорско време, па затоа една од нејзините типични примени е само за размена на клучеви кои ќе се користат за симетрична енкрипција (пораките се криптираат и декриптираат со истиот клуч) или пак за дигитално потпишување на документи.

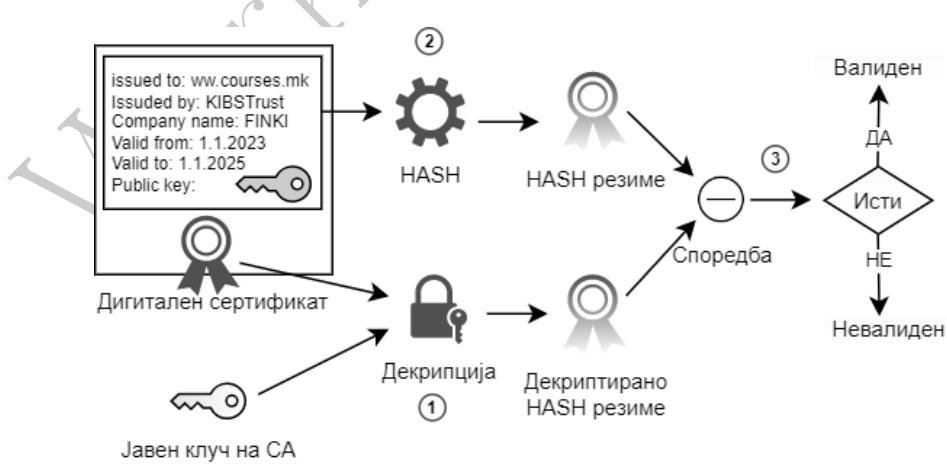
2. Барателот го зачувува приватниот клуч на безбедно место на веб серверот на кој подоцна ќе биде поставен и дигиталниот сертификат.
3. Барателот му го приложува неговиот јавен клуч на СА и документи со кој го докажува својот идентитет (лична карта, даночен број на компанијата и сл.). Во некои случаи, кога барателот не може физички да го докаже својот идентитет, особено ако станува збор за компанија која бара сертификат за веб сервер, тогаш само се испраќаат податоците за компанијата на барателот и во коренот на веб сајтот се поставува документ издаден од СА преку кој СА се уверува во постоењето на серверот и веб сајтот.
4. СА се уверува во веродостојноста на податоците и генерира дигитален документ кој содржи информации за барателот, валидност на документот, информации за самиот СА и јавениот клуч на барателот.
5. СА дигитално го потпишува документот со неговиот приватен клуч. Дигиталниот потпис се добива откако дигиталниот документ на барателот ќе се хешира и добиеното хеш резиме ќе се енкриптира со приватниот клуч на СА.
6. СА го прикачува дигиталниот потпис на дигиталниот документ и со тоа се добива дигитален сертификат за барателот, дигитално потписан од СА. Најчесто, како екstenзија на дигиталниот сертификат се користи . pem или . crt. СА му го предава дигиталниот сертификат на барателот.
7. Барателот го поставува сертификатот на веб серверот и го конфигурира сајтот да го користи протоколот HTTPS, наведувајќи ја локацијата на дигиталниот сертификат и на приватниот клуч.

Со оглед на тоа што јавниот клуч на СА е јавно достапен, било кој може да ја провери валидноста на дигиталниот сертификат. Процесот на валидација е прикажан на слика 1-8.

1. Валидаторот (веб прелистувачот) го извлекува дигиталниот потпис од дигиталниот сертификат и го декриптира со јавникот клуч на СА. Со тоа го добива оригиналното хеш резиме што го генерира СА за дигиталниот документ со податоци за корисникот.
2. Валидаторот го извлекува дигиталниот документ од дигиталниот сертификат и за него генерира хеш резиме.



3. Валидаторот ги споредува двете добиени хеш резимиња. Ако вредностите на двете низи се совпаѓаат, тогаш се потврдува дека податоците во дигиталниот сертификат се идентични со оригиналните податоци кои ги впишал CA кој го издал сертификатот.



Секако, валидаторот треба да му верува на CA дека е официјално тело кое

има право да издава сертификати и гарантира за секој издаден сертификат. За тоа, сите прелистувачи содржат листа на СА на кои им веруваат, па затоа, откако ќе го добијат сертификатот на барателот, го проверуваат дали е издаден од СА кој припаѓа на листата на прифатливи авторитети. Во зависност од тоа дали прелистувачот му верува на СА кој го издал сертификатот или не, пред самото поле за внесување на адреса прикажува соодветна ознака за да му даде до знаење на корисникот дали неговата комуникација е безбедна (најчесто е катанец за СА на кој му верува или прешкртан катанец за непознат СА). Сепак, доколку прелистувачот означи дека не му верува на СА, не значи дека комуникацијата е некриптирана, туку едноставно, не може да гарантира дека сопственикот на сертификатот е оној кој што тврди дека е (автентификација). Оваа ситуација е возможна ако корисникот самиот си генерира дигитален сертификат потпишан со клуч кој што самиот го генерира. Затоа, потребно е да сме особено претпазливи кога испраќаме чувствителни информации на веб сајтови кои користат HTTPS, но за кои прелистувачите алармираат дека сертификатот е издаден од СА кој не е во нивната листа на доверливи СА.

Откако дигиталниот сертификат ќе биде поставен на веб серверот, веб администраторот треба да го конфигурира веб серверот да слуша за TCP конекции на предефинираната HTTPS порта 443. Процесот на воспоставување на безбедна HTTP комуникација преку SSL се одвива во следните чекори (слика 1-9):

1. Корисникот креира барање за ресурс наведувајќи го протоколот HTTPS во URL адресата (<https://...>). Прелистувачот формира TCP конекција со серверот на предефинираната порта 443.
2. Прелистувачот испраќа листа на поддржани алгоритми за симетрична и асиметрична енкрипција.
3. Од понудените алгоритми, серверот ги враќа оние алгоритми кои тој ги поддржува.
4. Серверот го испраќа својот дигитален сертификат.
5. Прелистувачот го валидира сертификатот, а потоа генерира заедничка тајна која треба да ја сподели со серверот преку недоврливиот Интернет. Заедничката тајна ја користат и клиентот и серверот за да од неа генерираат заеднички клучеви за понатамошна симетрична енкрипција. Прелистувачот ја криптира тајната со јавниот клуч на серверот (кој го зема од неговиот дигитален сертификат).
6. Прелистувачот му ја испраќа на серверот криптираната тајна. Оваа порака може да биде декриптирана само со приватниот клуч на серверот, па затоа не постои никаква опасност дека пресретнатата криптирана порака ќе биде прочитана и декриптирана од потенцијален напаѓач.

7. Серверот, како сопственик на приватниот клуч, ја декриптира пораката и врз основа на споделената заедничка тајна го генерира истиот клуч за симетрична енкрипција како и клиентот.
8. Клиентот и серверот преговараат околу алгоритмот за симетрична енкрипција кој ќе се користи во понатамошната комуникација и разменуваат дополнителни параметри и заеднички клучеви за криптирање (за доверливост) и хаширање (за интегритет и автентикација) на пораките кои ќе се разменуваат.
9. Прелистувачот го криптира и хашира HTTP барањето и го испраќа до серверот преку TCP врската.
10. Серверот го декриптира HTTP барањето и го проверува интегритетот на пораката.

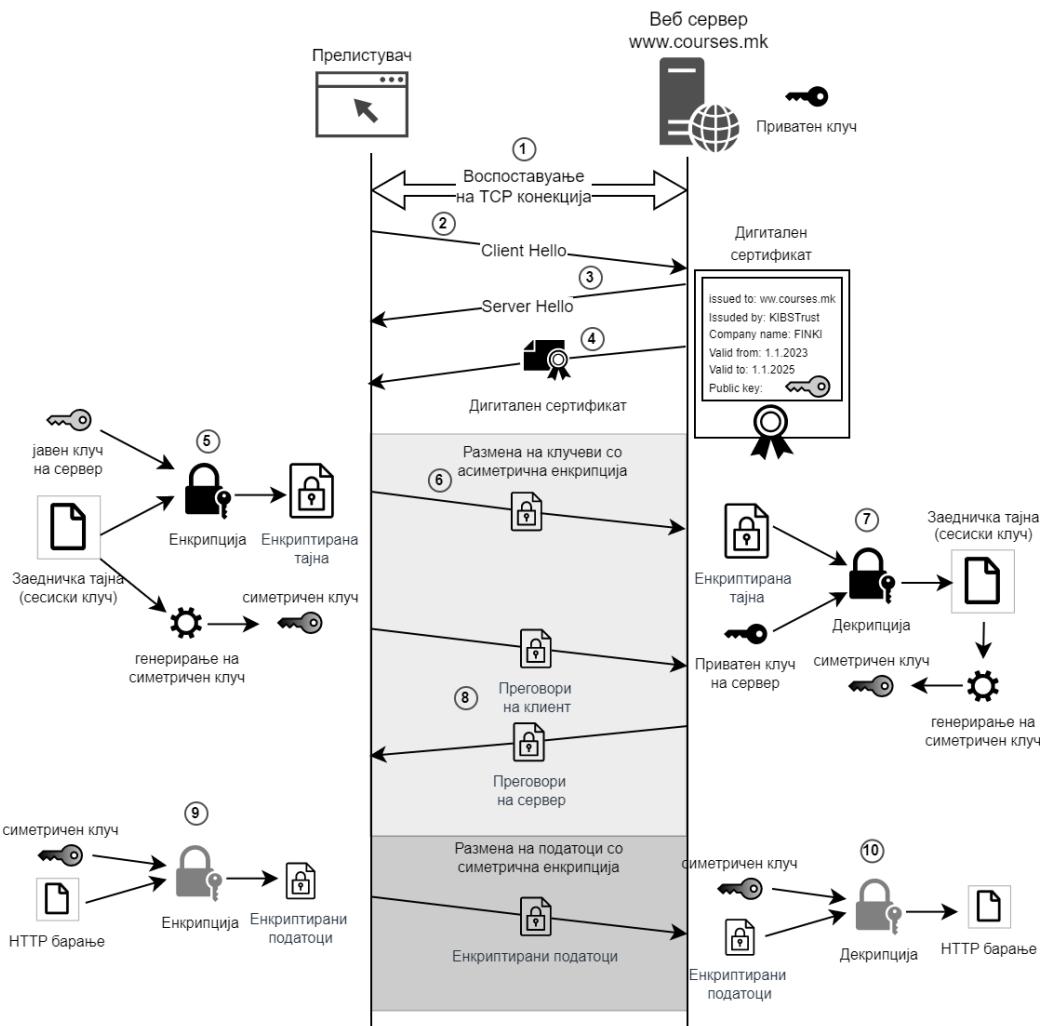
Целата понатамошна комуникација продолжува преку криптирани и хеширани пораки за кои се користат симетрични клучеви познати и на клиентот и на серверот, па затоа сите податоци од HTTP барањата и одговорите кои се разменуваат помеѓу нив се заштитени од надворешни напади. Сепак, и покрај тоа што заглавјето на барањето е заштитено, треба да се избегнува испраќање на сензитивни податоци во низата за пребарување од URL адресата бидејќи веб серверите најчесто ја запишуваат првата линија од HTTP барањето во логовите, во чиста текстуална верзија (некриптирана), па секој кој има пристап до серверот и логовите би можел да ги прочита овие податоци.

1.10 Примери за HTTP комуникација

Во процесот на развивање на веб апликации, често е потребно да се види содржината на барањата и одговорите кои ги разменуваат прелистувачот и веб серверот, времењата на достава, разменетите податоци и останати метрики за перформанси за целокупната комуникација. За таа цел, прелистувачите нудат напредни алатки во кои сите овие податоци можат да се разгледаат и анализираат на структуриран и прегледен начин. Алатката за развојни цели кај прелистувачот Chrome се нарекува DevTools и се стартува преку функцијското копче F12 или пак преку во менито со опции на прелистувачот. Слични алатки нудат и останатите познати прелистувачи.

Во продолжение ќе биде прикажана кратка анализа на посетата на неколку веб страници и размената на информации со веб серверот преку содржината на некои од поважните полиња на заглавјата и делови на барањата и одговорите.

На сликата 1-10 е прикажана комуникацијата помеѓу клиент и сервер која започнува откако клиентот кликнува на линк на веб страница. Линкот претста-

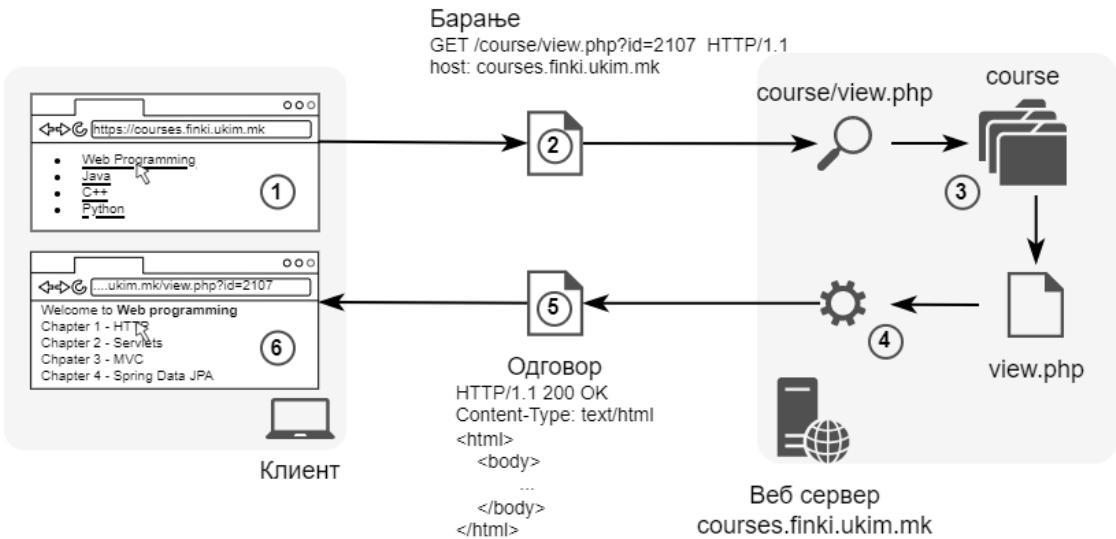


Слика 1-9: Воспоставување на SSL врска.

вуша референца кон друг ресурс, па затоа прелистувачот генерира GET барање за ресурс кој преставува скрипта која се извршува на серверска страна (.php). Веб серверот го добива барањето, го бара ресурсот во својот податочен систем, ја извршува скриптата која како резултат дава HTML код и генерира HTTP одговор кој го испраќа до клиентот. На крај, прелистувачот ја обработува добиената содржина и ја прикажува на екранот.

Содржината на барањето кое е испратено како резултат на кликување на линк кој е дел од HTML страница добиена од сајтот <https://courses.finki.ukim.mk> е дадена во изворниот код 1.1.

```
GET /course/view.php?id=2107 HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,
image/webp,image/apng,*/*;q=0.8
```



Слика 1-10: Пример за HTTP комуникација помеѓу клиент и сервер со GET барање

```

Accept-Encoding: gzip, deflate, br
Accept-Language: en-GB,en;q=0.9,mk;q=0.7,es;q=0.6,it;q=0.5
Cache-Control: no-cache
Connection: keep-alive
Cookie: _ga=GA1.2.2049884469.1512582575; SRVNAME=COURSES4;
        MoodleSession=baq34517catud5cbt3r2sqqocm8
Host: courses.finki.ukim.mk
Referer: https://courses.finki.ukim.mk/
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/106.0.0.0 Safari/537.36

```

Изворен код 1.1: Содржина на HTTP GET барање

Според првата линија, прелистувачот генерира GET барање за ресурс со релативна локација </course/view.php?id=2107>. Ресурсот се наоѓа на сервер чие име може да се прочита од полето Host: courses.finki.ukim.mk. Важно е да се забележи дека првата линија од барањето не ја содржи апсолутната адреса на ресурсот. Иако во податоците прикажани во DevTools може да се идентификува IP адресата на домаќинот, таа не е дел од заглавјето и прелистувачот ја добива од сокетот кој го користи за да ја воспостави TCP врската со серверот.

Од релативната адреса на ресурсот може да се забележи дека клиентот испраќа дополнителни податоци кон серверот преку низата на пребарување (query string, англ.). Поточно, станува збор за параметар со име id со вредност 2107 кој му е потребен на бараниот ресурсот, односно скриптата на локација </course/view.php> за да генерира сооветен HTML кој ќе биде испратен како

одговор.

ПРЕЛИСТУВАЧОТ ја поддржува верзијата HTTP/1.1, што е наведено како информација во првата линија од барањето.

Барањето за ресурсот потекнува од страница со URL <https://courses.finki.ukim.mk/>, што може да се види од полето **Referer**. Тоа значи дека пред да го креира барањето, корисникот клика на некој линк кој се наоѓал на страницата од полето **Referer**.

Барањето содржи и параметри кои прелистувачот ги користи за да преговара за преференците околу содржините кои ќе ги размени и начинот на кој ќе комуницира со серверот. Според заглавјето **Connection**, клиентот назначува дека поддржува да ја остави TCP конекцијата отворена (**keep-alive**), за да преку неа ги побара и останатите ресурси кои ќе бидат дел од ресурсот кој ќе го добие.

Во однос на поддржаните содржини, според полето **Accept**, клиентот преферира текстуални содржини, поточно HTML и HTML+XML со максимален тежински фактор (кој изнесува 1), потоа XML со тежински фактор 0.9, слики од типот AVIF, WEBP и APNG со максимален тежински фактор и сите останати MIME типови со тежински фактор 0.8.

ПРЕЛИСТУВАЧОТ поддржува три алгоритми за компресија на содржината на одговорот наведени во **Accept-Encoding** и тоа **gzip**, **deflate** и **br** со подеднаква префериранца за кој било од нив. Доколку серверот го има ресурсот на повеќе јазици, тогаш прелистувачот преферира да добие содржина на британски англиски јазик (**en-GB**), која било останата верзија на англиски (**en**) со тежински фактор 0.9, а потоа македонски, шпански и италијански јазик со фактор 0.7, 0.6 и 0.5, соодветно.

Полето **User-Agent** прелистувачот го прецизира неговиот контекст, односно наведува дека станува збор за верзијата 106.0.0.0 на прелистувачот Chrome кој е инсталiran на Windows NT 10.0 оперативен систем. Иако во листата на софтвер стојат Mozilla, Safari и AppleWebKit, тоа не значи дека тие се дел од контекстот на прелистувачот, туку дека прелистувачот е компатибilen со сите нив. Ваквата опширна листа знае често да е конфузна и двосмислена поради големиот број на наведени софтвери, но постојат правила според кои може точно да се прецизира контекстот. Причината има историски корени, што е надвор од целите на книгата.

Полето **Cookie** укажува дека корисникот веќе пристапил до сајтот и од него добил колачиња кои прелистувачот ги зачувал во својата меморија и сега ги испраќа во секое барање. Прелистувачот испраќа три колачиња од кои дел се користат за идентификација на корисничката сесија по успешна најава. Повеќе информации за нивната големина, животен век и домен на важење може да се видат во јазичето Cookies на алатката DevTools.

Полето `Cache-Control` со вредност `no-cache` означува дека клиентот бара најнова верзија на ресурсот, иако можеби веќе има кеширано некоја претходна верзија.

Дел од одговорот кој го добил прелистувачот од серверот како резултат на барањето од изворниот код 1.1 е даден во изворниот код 1.2.

```
HTTP/1.1 200 OK
Date: Wed, 12 Oct 2022 12:05:55 GMT
Server: Apache/2.4.38 (Debian)
Expires: Mon, 20 Aug 1969 09:23:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Content-Language: en
Cache-Control: post-check=0, pre-check=0, no-transform
Last-Modified: Wed, 12 Oct 2022 12:05:55 GMT
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 22854
Connection: close
Content-Type: text/html; charset=utf-8
```

Изворен код 1.2: Содржина на одговор на HTTP GET барање

Серверот одговара дека ја поддржува истата верзија на протоколот HTTP како и клиентот (HTTP 1.1). Статусот на барањето е успешен, со код 200 и опис на статусот `OK`.

Полето `Date` го означува временскиот момент кога е креиран одговорот, а полето `Server` значи дека одговорот го генерира веб сервер од типот Apache, верзија 2.4.38.

Датумот во далечното минато во полето `Expires` означува дека рокот на одговорот е веќе поминат, што му дава до знаење на прелистувачот да не ја кешира содржината за следна употреба. `Last-Modified` во случајов има иста вредност како и `Date` и му служи на прелистувачот да знае кога последен пат е променет ресурсот. Доколку прелистувачот има кеширана верзија на истиот ресурс, ова поле му помага да види дали добиената верзија е понова, со цел да ја замени старата со новата верзија. Со оглед на тоа што содржината веќе има истечен рок, ова поле не е од полза за прелистувачот во конкретниот пример.

Во однос на кеширањето, серверот веќе му ги наведува правилата на прелистувачот во полето `Cache-control`. Со `no-store` му укажува да не ја зачувува содржината во локалниот кеш, со `no-cache` му укажува дека ако веќе има кеширано содржина, треба да му испрати ново барање за ревалидација пред да ја реискористи за приказ, а со `must-revalidate` му укажува да направи ревалида-

ција само ако страницата не е со поминат рок. Ова е типична група на ознаки кои се користат како вредности на полето **Cache-control**, кога целта е да оневозможи кеширање на клиентска страна (или да се сведе на минимум).

Ресурсот кој го испраќа серверот во телото на одговорот е со големина од 22854 байти, што може да се види во полето **Content-Length**. Овој податок му укажува на прелистувачот дека после празната линија на одговорот треба да исчита вкупно 22854 байти. Испратената содржина е компресирана и не може директно да се употреби, па за да ја декомпресира, прелистувачот треба да го употреби алгоритмот **gzip**, наведен во полето **Content-Encoding**. По декомпресијата, прелистувачот треба да ги третира добиените податоци како HTML содржина чии знаци се кодирани според UTF-8 кодната шема, што може да се види од полето **Content-Type** со MIME вредност **text/html; charset=utf-8**. Јазикот на содржината во одговорот е англиски (**Content-Language: en**). При користење на DevTools, конкретната HTML содржина може да се погледне во јазичето Response.

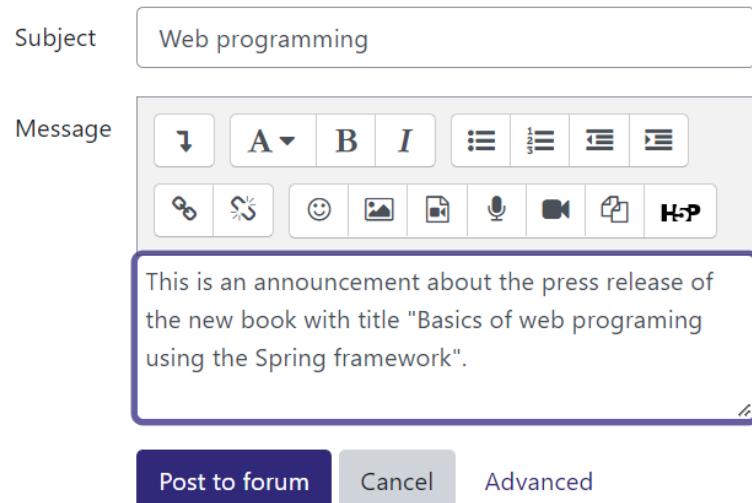
Иако клиентот во барањето преферира конекцијата да остане отворена, со вредноста **close** на полето **Connection** серверот му укажува дека по достава на содржината тој ќе ја затвори TCP врската.

Во продолжение ќе разгледаме пример каде корисникот испраќа POST барање и ќе ја анализираме содржината на разменетите пораки помеѓу серверот и клиентот, задржувајќи се само на полињата кои се од важност и специфични за конкретната размена.

На сликата 1-11 е даден изгледот на дел од HTML страница која содржи форма со веќе пополнети полиња за објавување на тема на форум и копче кое го поднесува барањето до серверот. Формата содржи поле за наслов, поле со HTML уредувач за внесување на содржина и копче за снимање на содржината. Дополнително, формата содржи и невидливи елементи кои служат за идентификација на курсот на кој ќе се објави темата и нишката од теми на која ќе припаѓа.

Редуцираната и најосновна HTML репрезентацијата на формата и нејзините визуелни и скриени елементи, без дополнителни класи за изгледот, е дадена во изворниот код 1.3. За да се добие HTML кодот на целокупната страница со фокус на даден елемент, се користи опцијата Inspect врз елементот од интерес (десен клик и избор од менито) во самиот прелистувач.

```
<form method="POST" action="">
    <input type="hidden" name="course" value="2107" />
    <input type="hidden" name="forum" value="3214" />
    <input type="text" name="subject" />
    <textarea name="message[txt]">
```



Слика 1-11: Пример за HTML форма за генерирање на POST барање

```
<input type="submit" name="submitbutton" value="Post to forum" />
</form>
```

Изворен код 1.3: HTML репрезентација на форма која генерира HTTP POST барање

Аргументот `method` на формата означува дека копчето со улога `type="submit"` ќе генерира POST барање во чие тело ќе се испратат податоците од влезните елементи кои ги содржи, а барањето ќе биде обработено од ресурсот назначен преку аргументот `action`, во конкретниот случај, од скрипта со URL <https://courses.finki.ukim.mk/mod/forum/post.php>.

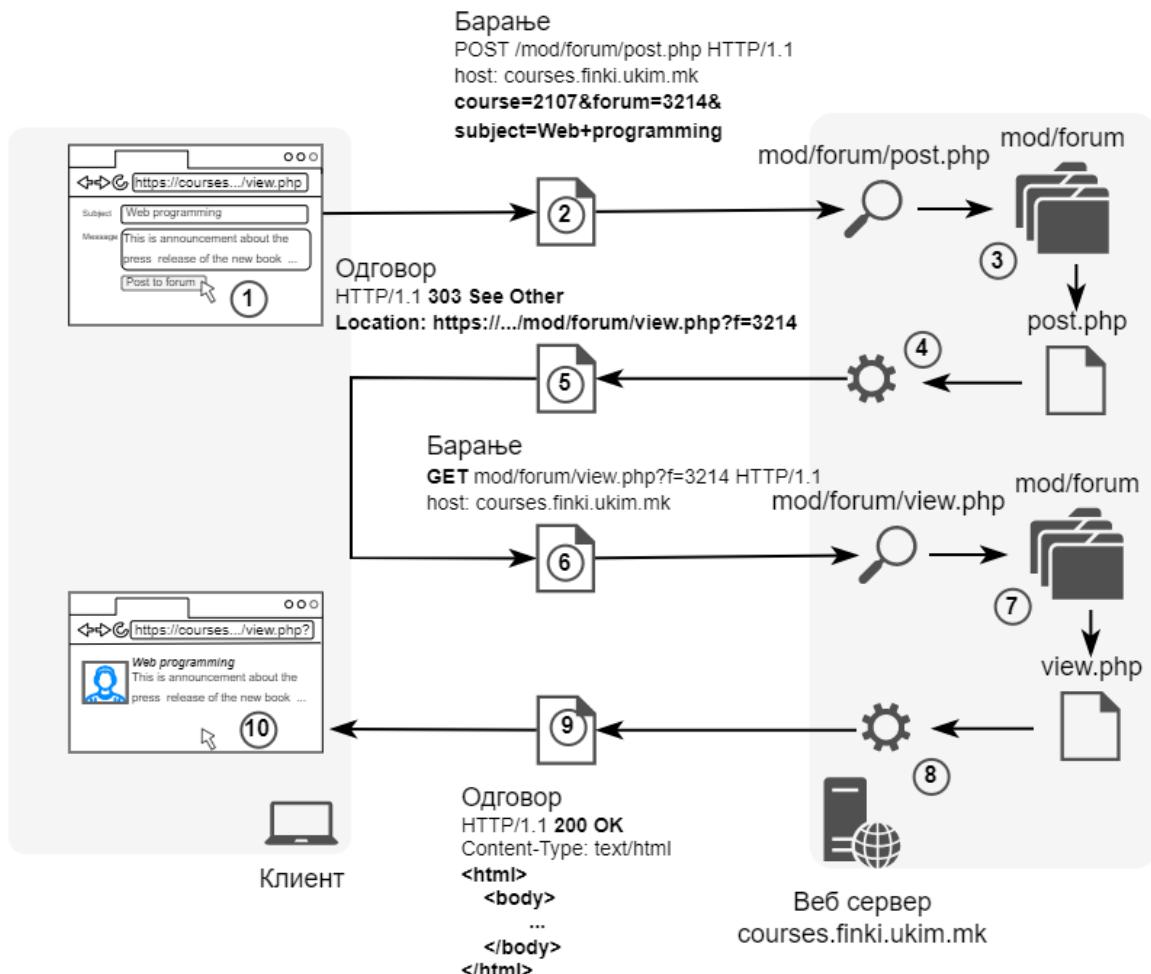
На сликата 1-12 е прикажана целокупната комуникацијата помеѓу клиентот и серверот која започнува откако клиентот кликнува на копчето за зачувување на темата. Прелистувачот генерира POST барање за ресурс кој претставува скрипта (.php) за зачување на податоците пренесени во телото (чекори 1-2).

Содржината на дел од HTTP барањето кое го генерира формата е прикажана во изворниот код 1.4.

```
POST /mod/forum/post.php HTTP/1.1
...
Content-Length: 460
Content-Type: application/x-www-form-urlencoded
...
```

Изворен код 1.4: Содржина на HTTP POST барање

Во првата линија, барањето го содржи методот POST, релативната адреса



Слика 1-12: Пример за HTTP комуникација помеѓу клиент и сервер со POST барање

на ресурсот кој треба да го обработи барањето и верзијата на протоколот. Може да се забележи дека релативната адреса `/mod/forum/post.php` е истата како и содржината на аргументот `action` на формата. Првиот дел од адресата, односно хостот е во полето `Host` кое е намерно изоставено во примерот. POST барањето содржи податоци во неговото тело со големина назначена во полето `Content-Length`, што значи дека почнувајќи од празната линија после заглавјето, серверот ќе прочита 460 бајти. Според полето `Content-Type` чија MIME вредност е `application/x-www-form-urlencoded`, серверот ќе знае дека овие податоци треба да ги третира како URL-кодирани податоци од форма.

URL-кодираната содржина на телото како и нејзината парсирана верзија се прикажани во кодниот сегмент 1.5. За да се погледне содржината на телото во алатката DevTools потребно е да се избере јазичето `Payload`. Иако телото е дел од барањето, за поголема прегледност, алатката го прикажува во посебно

прозорче од она во кое се прикажани заглавјата.

```
\URL encoded (original)
course=2107&forum=3214&subject=Web+programming&message%5Btext%5D=%3Cp+dir...
This+is+an+announcement+...+the+Spring+framework%22.%3C%2Fp%3E
\Parsed
course: 2107
forum: 3214
subject: Web programming
message[text]: <p dir="ltr" style="text-align: left;">This is an
    announcement ... the Spring framework".</p>
```

Изворен код 1.5: URL-кодирана и парсирана содржина на тело на HTTP POST барање

Ако се споредат двете верзии на содржината и HTML влезните елементи, ќе се забележи дека имињата на елементите се користат како клучеви на податоците кои се пренесуваат во форма на **клуч=вредност** парови одвоени со &. Алфаниумеричките знаци се непроменети, додека специјалните знаци се кодирани, на пример, знакот празно место е кодиран со + (според URL кодирањето и %20 е валиден код), знакот < е кодиран со %3C, итн. Веб серверот го добива барањето, го бара ресурсот во својот податочен систем и ја извршува скриптата која како резултат генерира HTTP одговор за пренасочување (чекори 3-5 на слика 1-12).

Одговорот кој го испратил серверот по процесирање на POST барањето е прикажан во изворниот код 1.6. Статусот на одговорот е 303 со статусен опис **See Other**. Овој код припаѓа на фамилијата кодови за пренасочување и во конкретниот случај означува дека бараната скрипта ги обработила пратените податоци, односно успешно ја снимило темата, но за да се добие крајниот изглед во кој ќе биде прикажана зачуваната тема, потребно е прелистувачот да генерира барање кон нов ресурс кој има URL со вредност <https://courses.finki.ukim.mk/mod/forum/view.php?f=3214>, наведена во полето **Location**. Статусниот код 303 често се користи како одговор на POST или PUT методи после снимање или ажурирање на податоци.

```
HTTP/1.1 303 See Other
...
Location: https://courses.finki.ukim.mk/mod/forum/view.php?f=3214
...
```

Изворен код 1.6: Содржина на одговор на HTTP POST барање

Како реакција на овој одговор, во позадина, прелистувачот генерира ново GET барање кон посочениот ресурс, без никаква интервенција од страна на корисникот (чекор 6 на слика 1-12). Неговата содржина е дадена во изворниот код 1.7

```
GET /mod/forum/view.php?f=3214 HTTP/1.1
```

```
...
```

Изворен код 1.7: Содржина на HTTP GET барање како одговор на пренасочување

Конечно, серверот го прима барањето и откако ќе го најде бараниот ресурс, генерира одговор со статус 200 OK, кој во заглавјето посочува дека телото носи компресирана HTML содржина со големина од 41721 бајти, по што прелистувачот ја прикажува во прозорецот (чекори 7-10 на слика 1-12). Дел од содржината на одговорот е прикажан во изворниот код 1.8.

```
HTTP/1.1 200 OK
```

```
...
```

```
Content-Encoding: gzip
```

```
Content-Length: 41721
```

```
Content-Type: text/html; charset=utf-8
```

```
...
```

Изворен код 1.8: Содржина на одговор на HTTP GET барање

Глава 2

Основи на J2EE

2.1 Што е JEE и зошто е потребна?

Постојат повеќе Јава платформи во зависност од типот на апликациите за кои се наменети. Сите платформи се состојат од JVM¹ и колекција од софтверски компоненти кои се користат за развивање апликации, т.н. апликациски програмски интерфејси (API). Основната Јава платформа се нарекува JABA стандардно издание (Java Standard Edition (JSE), англ.) и содржи основни типови и класи за Јава програмскиот јазик, но и понапредни класи за вмрежување, безбедност, пристап до бази на податоци и развивање на графички интерфејси.

Јава серверското издание (Java Enterprise Edition (JEE), англ.²) е платформа за развивање на повеќеслојни, скалабилни, безбедни и доверили деловни веб апликации кои се извршуваат на сервер. Претставува проширување на JSE кое, меѓу другото, воведува компоненти како што се сервлети (servlets, англ.) и контейнери (containers, англ.). Со тоа JEE значително го олеснува развивањето на сложени веб апликации, бидејќи ги сведува на збир од стандардизирани компоненти на кои им обезбедува низа на сервиси, криејќи ја од програмерите целата комплексност на нивната имплементација.

Во глава 1, при воведување на концептите на WWW, го опишавме процесот на обработка на барања за динамички веб ресурс од страна на веб серверите. При тоа, нагласивме дека Веб серверот мора да има дополнителен софтвер кој ќе може динамички да ја генерира содржината на страниците во зависност од испратените податоци. Кога станува збор за Јава, оваа функционалност ја обез-

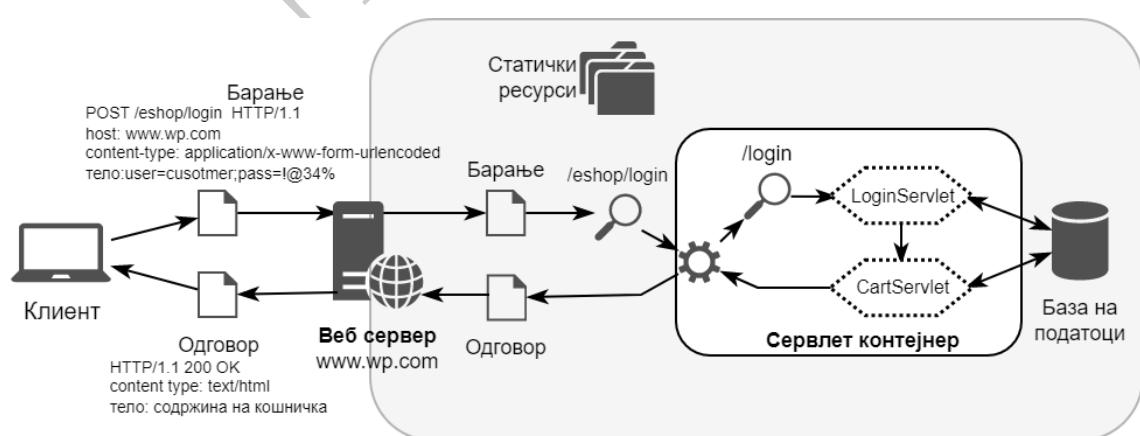
¹Јава виртуелната машина му овозможува на компјутерот да извршува Java програми, како и програми напишани на други јазици кои исто така се компајлирани во Java бајткод. JVM е детално описана со спецификација која формално опишува што е потребно во имплементацијата на JVM.

²Серверско издание, затоа што неговата намена не е исклучиво за компании/кропорации, туку за сите веб апликации кои се поставуваат на сервер

бедуваат сервлет контејнерот и сервлетите кои се едни од основните компоненти на JEE.

Сервлетите се Java класи со имплементација на бизнис логиката потребна за генерирање на динамички страници и се повикуваат во зависност од побараната URL адреса. Имплементацијата на сервлетите мора да ги почитува спецификациите за Java сервлети и вклучува имплементација на дефинирани методи кои се извршуваат во текот на нивниот животен циклус. Но, сервлетите немаат `main()` метод за нивно извршување. Тие се контролирани од страна на сервлет контејнерот. Сервлет контејнер пак е Java позадински процес кој се ивршува од страна на JVM и чија главна задача е да ги пресликува барањата и одговорите во соодветни објекти и да управува со животнот циклус на сервлетите и останатите компоненти кои се дел од веб апликацијата.

Сервлет контејнер има задача да го прифати пристигнатото барање од веб серверот и да го предаде на соодветен сервлет. Секоја веб апликација се состои од повеќе сервлети и секој од нив е одговорен за имплементација на одредена функционалност. На слика 2-1 е прикажана комуникацијата помеѓу клиент и веб апликација за е-трговија која треба да ги прикаже продуктите кои корисникот ги сместил во неговата кошничка. Разгледуваме сценарио каде при првиот пристап се појавува прозорец во кој корисникот треба да внесе корисничко име и лозинка. Во зависност од тоа што е внесено, во прелистувачот се појавува порака за грешка, во случај на неуспешна најава, или пак страница со динамички генерирана листа на продукти од потрошувачката кошничка чија содржина зависи од историјата на претходните активности на корисникот. Веб серверот ги прифаќа и разбира HTTP барањата од прелистувач, но за да се опслужват барањата во целост, веб серверот му ги проследува барањата на сервлет контејнерот и од него очекува динамички генерирана содржина.



Слика 2-1: Опслужување на барање за динамички веб сайт (апликација)

Сервлет контејнерот го конвертира барањето во објект и го проследува до соодветниот сервлет кој ги извлекува податоците за корисничко име и лозинка и ја споредува вредноста на лозинката од барањето со вредноста на лозинката за тоа корисничко име сместена во базата на податоци. Доколку ове две вредности се совпаѓаат, ги чита од базата сите продукти од потрошувачката кошничка зачувани за тоа корисничко име, генерира соодветна HTML содржина за приказ на продуктите и таа содржина ја вметнува во објект кој го репрезентира одговорот. Потоа, сервлетот му го предава одговорот на сервлет контејнерот, кој го трансформира одговорот во соодветен HTTP формат и му го испраќа на веб серверот за достава до клиентот. Според тоа, контејнерот е посредник помеѓу веб серверот и сервлетите. Тој повикува сервлети кои ја имплементираат бизнис логиката за динамичко креирање на веб страници.

2.2 Сервлет контејнер

Сервлет контејнерот е одговорен за менаџирање на сервлетите но и нуди дополнителни услуги:

- **Поддршка на комуникацијата:** Сервлет контејнерот овозможува лесна комуникација со веб серверот и со клиентот. Така, нема потреба од креирање и отворање на сокети (`ServerSocket`), слушање на порти и креирање на влезни и излезни текови (`InputStream`, `OutputStream`). Целата оваа комплексност е сокриена зад добро дефинирани апликациски програмски интерфејски, за да можеме да се фокусираме на имплементација на бизнис логиката во сервлетите.
- **Управување со животните циклуси:** Сервлет контејнерот се грижи за животните циклуси на сервлетите, како што се нивното креирање, иницијализација, повик и терминирање. На овој начин, програмерите не мора да се грижат за менаџирање на ресурсите, бидејќи ова е задача на сервлет контејнерот.
- **Паралелизација:** Сервлет контејнерот овозможува паралелно процесирање на повеќе барања. За оваа цел, се креира посебна нишка за процесирање на секое барање.

Секоја веб апликација од јава серверското издание се состои од повеќе менаџирани компоненти:

- **Сервлетите** (`Servlet`) се компонентите кои ја содржат **бизнис логиката** за креирање на одговор од добиените барања. За секое барање, сервлет контејнерот повикува најмногу еден сервлет.

- **Филтрите** (Filter) се компоненти кои ги **пресретнуваат** барањата и вршат модификации на барањата и одговорите, пред истите да бидат проследени на сервлетите. За секое барање, контејнерот може да повика повеќе филтри, пред да ја додели контролата на сервлетот.
- **Обсерверите** (Listener) се компоненти преку кои може да се **регистрираме за следење на настани** кои се генерираат во рамките на сервлет контејнерот и да дефинираме соодветна реакција кога ќе се случат.

Сите овие компоненти се јава класи кои имплементираат соодветни интерфејси од програмскиот интерфејс на јава серверското издание (JEE API) и кои се регистрираат во дескрипторот за поставување. **Дескрипторот за поставување** е логичка репрезентација на сите компоненти кои може да бидат повикани при процесирање на барањата. Оваа логичка репрезентација може да се конфигурира со користење на XML (eXtensible Markup Language) синтакса во `web.xml` или со поставување на соодветни анотации на самите менаџирани компоненти. Менаџираните компоненти ја содржат бизнис логиката на апликацијата, па по нивната имплементација, заедно со останатите ресурси се пакуваат како веб архиви (**war**).

На слика 2-2 е прикажан живонтиот циклус на една веб апликација составена од менаџирани компоненти:

1. Програмерот креира проект според JEE спецификацијата во својата интегрирана околина за развој (IDE) инсталрирана на неговиот компјутер кој ќе го користи за развој. Постојат голем број на IDE, но една од најпогодните за развивање на Јава веб апликации е IntelliJ. Во овој чекор, самата околина креира соодветна структура од директориуми и предефинирани датотеки кои се неизбежен дел од веб апликација.
2. Програмерот креира менаџирани компоненти како Јава класи во кои ја имплементира целокупната бизнис логика на веб апликацијата.
3. Програмерот ги регистрира менаџираните компоненти во Дескрипторот за поставување (`web.xml`) или преку анотации директно во самите класи. На slikата е прикажан комбиниран начин на регистрирање на компонентите.
4. По имплементација на посакуваната бизнис логика, програмерот ја компајлира и запакува веб апликацијата. Како резултат на овие операции се добива веб архив со екstenзија (**war**). Оваа архива преставува датотека која го содржи целиот код потребен за веб апликација и е единствена датотека која се користи за нејзино поставување.
5. Програмерот ја предава **war** датотеката на DevOps инженерот за да ја постави на продукцискиот веб сервер на кој, покрај апликација за веб сервер, има инсталрирано JRE и сервлет контејнер кој ги имплементира JEE

спецификациите (во примерот, Apache Tomcat). Сервлет контејнерот функционира како дел од веб серверот. Во рамките на еден веб сервер, може да бидат поставени (deployed) повеќе веб апликации. Поставувањето најчесто се прави со мануелно копирање³ на архивата на соодветна локација дефинирана за веб серверот⁴.

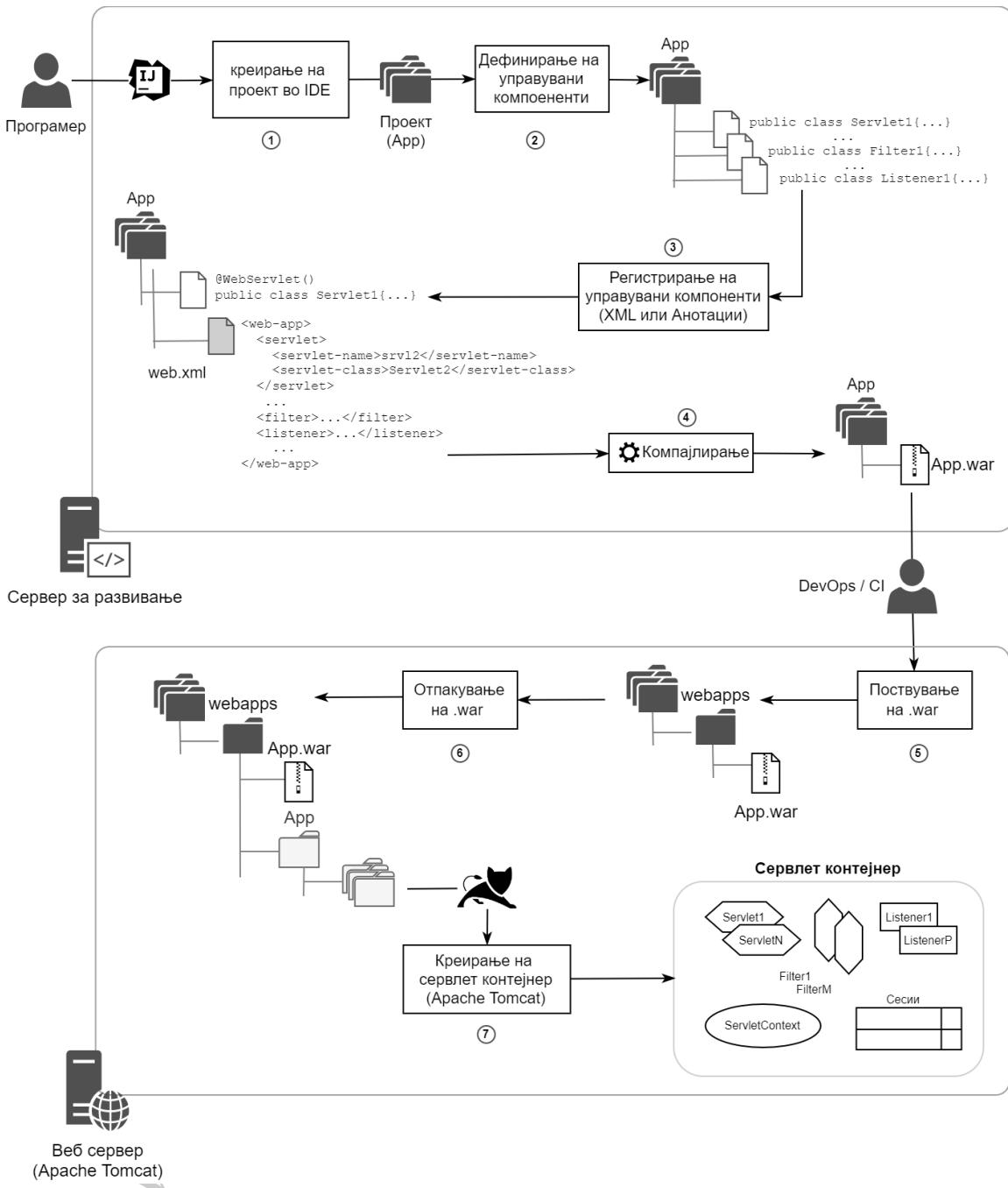
6. Архивата се отпакува и со тоа се добива веб апликација организирана во датотечната структура од именици
7. Веб серверот креира сервлет контејнер за апликација. За секоја од поставените веб апликации, веб серверот иницијализира посебна инстанца од **сервлет контејнерот**. На овој начин се овозможува изолирано извршување на различните апликации, без дополнително делење на ресурси. Потоа, сервлет контејнерот го вчитува дескрипторот за поставување⁵ и започнува со неговиот животен циклус. Во овој момент, сервлет контејнерот ги регистрира сите менацирани компоненти според конфигурацијата во дескрипторот за поставување. Најопшто гледано, **дескрипторот за поставување** ги дефинира **иницијализиските параметри** за секоја од **менацираниите компоненти**, како и нивно **мапирање со URL патеките на барања** за кои тие треба да се селектираат. Дополнително, сервлет контејнерот креира посебен споделен мемориски простор за сите компоненти т.н. **ServletContext** како и структури за чување на **сесиски податоци** за корисниците. По овој чекор, сервлет контејнерот е спремен за опслужување на HTTP барања.

Откако ќе се постави веб апликацијата, таа започнува со обработка на HTTP барања кои ѝ ги проследува Веб серверот. **Веб серверот** е задолжен за **процесирање на пораките од HTTP протоколот**. Ова значи дека веб серверот е одговорен за **отворање на сокет** (`ServerSocket`) кој слуша на конфигурирана **порта**. Штом ќе биде добиена порака од страна на сокетот, тој ги исчитува HTTP барањата. Врз основа на побараната патека, **веб серверот го идентификува соодветниот сервлет контејнер**, и го делегира HTTP барањето кон него. Во овој процес, веб серверот ги конвертира првата линија и заглавјата од HTTP барањето во текст, а за телото креира влезен тек (`InputStream`). Истовремено креира и излезен тек (`OutputStream`) кој е наменет за изградба на одговорот. На крај, ги делегира текстот за HTTP барањето, влезниот тек и излезниот тек на селектираниот сервлет контејнер. Од овој момент, иницијализираниот сервлет контејнер

³Може да се користи и автоматизирано поставување со користење на софтвер за континуална интеграција (Continuous Integration англи.)

⁴За Apache Tomcat, тоа е фолдерот `${TOMCAT_ROOT}/webapps`

⁵Ако е присутен `web.xml`, се исчитува неговата содржина, по што се продолжува со скенирање на класите и процесирање на нивните JEE анотации.



Слика 2-2: Животен циклус на еден проект

продолжува со процесирање на барањето, со селекција на филтрите и сервлетот за тоа барање и нивно повикување во посебна нишка. Притоа, контејнерот од делегираните влезни податоци ги креира `HttpServletRequest` и `HttpServletResponse` објектите и истите ги делегира на селектирани филтри и сервлет. Филтрите и сервлетот ја вметнуваат потребната содржина во излезниот тек на одговорот според имплементираната бизнис логика. По завршувањето на филтрите и сер-

влетот, веб контејнерот го враќа одговорот на веб серверот, кој ги структуира соодветните податоци според HTTP протоколот во HTTP одговор и го испраќа истиот кон клиентот.

2.3 Сервлети

2.3.1 Имплементација

Сервлетите ја содржат бизнис логиката на една веб апликација. Во општ случај, за да се имплементира бизнис логиката на еден сервлет, потребно е да се креира класа која ќе ја проширува апстрактната класата `javax.servlet.GenericServlet`.⁶ Класата `GenericServlet` го имплементира интерфејсот `Servlet`, кој ги содржи методите `init()`, `service()` и `destroy()`. Таа има минимална имплементација која не зависи од комуникацискиот протокол и само го олеснува креирањето на сервлети преку проширување на методите од интерфејсот `Servlet`. На пример, за да се добие функционален сервлет, потребно е да се препокрие методот `service()`.

```

1 import javax.servlet.*;
2 public class FirstGenericServlet extends GenericServlet{
3
4     public void service(ServletRequest request, ServletResponse response) {
5         ...
6     }
7 }
```

Изворен код 2.1: Креирање на генерална сервлет класа

Сепак, за пишување на веб апликации со кои клиентите комуницираат преку протоколот HTTP, потребен е сервлет кој ќе ја разбира структурата на HTTP барањата и одговорите и со тоа ќе го олесни пристапот до содржината на барањето и генерирањето на одговорите при развивање на веб апликации. Затоа, во најголем број од случаите, бизнис логиката на сервлетите се имплементира преку проширување на апстрактната класа `HttpServlet`, која сама по себе е екstenзија на `GenericServlet`.

```

1 public class FirstHttpServlet extends HttpServlet{
```

⁶Класата е дел од пакетот `javax.servlet`, па затоа и мора експлицитно да се вклучи за да биде препознана. Оваа класа, како и сите класи кои се дел од оваа глава припаѓаат на пакетот `javax.servlet`, па можат да се вклучат или поединечно или пак да се вклучи целиот пакет со `import javax.servlet.*`. Во остатокот од главата, `import` делот ќе биде намерно изоставен во примерите, а за вклучување на соодветните класи од овој ќе ви помогне вашата IDE алатка.

```
2     doGet(HttpServletRequest request, HttpServletResponse response){  
3     }  
4     doPost(HttpServletRequest request, HttpServletResponse response){  
5         ...  
6     }  
7     ...  
8 }
```

Изворен код 2.2: Креирање на сервлет класа која може да опслужува HTTP GET и POST барања

Класата ги дефинира методите `doGet()`, `doPost()`, `doPut()`, `doDelete()`, `doHead()`, `doOptions()` и `doTrace()` кои се повикуваат од `service()` методот, во зависност од методот на HTTP барањето. Како аргументи на секој од овие методи се пренесуваат инстанци од објектите `HttpServletRequest` и `HttpServletResponse`. `HttpServletRequest` содржи својства кои ги репрезентираат соодветните елементи на HTTP барањето. `HttpServletResponse` содржи својства со сите елементи на HTTP одговорот. На овој начин, со помош на објектно-ориентиран пристап може да се манипулира со барањето (да се читаат вредности) и одговорот (да се запишуваат вредности).

Ако се навратиме на иницијалниот пример за најава на е-продавница каде треба да имплементираме најава со корисничко име и лозинка, имплементацијата можеме да ја направиме преку креирање на сервлет `LoginServlet`. Бидејќи корисничкото име и лозинката се сензитивни приватни информации, тие се испраќаат во телото на HTTP барање од типот POST. Затоа, во сервлетот за најава треба да се имплементира методот `doPost()` кој треба да ја провери валидноста на лозинката и да креира содржината на повратната порака преку нејзино вметнување во објектот за одговорот. Откако ќе заврши методот `doPost()`, контејнерот го составува HTTP одговорот и го враќа на клиентот. Според тоа, целата имплементација се сведува само на пишување на бизнис логика на објектно-ориентирано ниво без да се води грижа за HTTP комуникацијата. Развивањето на веб апликацијата би било значително покомплексно без сервлети бидејќи секое HTTP барање испратено од клиентот ќе треба да се обработува како низа од знаци во која треба најпрво да се лоцира URL адресата на барањето, потоа типот на метод според кој ќе се повика некој метод за обработка, а потоа, во тој метод да се лоцира името на параметарот (на пример `username`) и да се генерира HTTP одговорот како низа од знаци која ќе му се предаде на веб серверот. Веб серверот и сервлет контејнерот значително го олеснуваат развојот на веб апликации и овозможуваат интерфејс преку кој се фокусираме на бизнис логиката која треба да се имплементира.

2.3.2 Регистрација на сервлети и пресликување

За да може сервлет контејнерот да управува со сервлетите и да одлучува кој сервлет да го повика при обработка на секое барање, потребно е сервлетите и нивните пресликувања да се регистрираат во Дескрипторот за поставување или во самите класи со помош на анотации. Кога станува збор за конфигурација во дескрипторот за поставување, тогаш, за секој сервлет, во датотеката со име `web.xml` сместена во именикот `WEB-INF/` се дефинираат следните елементи:

- **Име на сервлет:** Се користи како име на инстанца која ќе се изведе од имплементациската класа. Треба да биде уникатно и служи за интерна употреба на програмерот при негово пресликување со URL адреса. Името не мора да биде поврзано ниту со името на класата, ниту со URL адресата и може да се менува без последници врз однесувањето на апликацијата.
- **Име на класа:** Претставува име на класата во која е имплементирана неговата бизнис логиката и
- **Шаблон за URL адреса:** дефиниција на правила во кои се поставува за кои URL адреси на барањето ќе биде повикан сервлетот.

Податоците за име на сервлет и имплементациска класа се дефинираат во елементот `<servlet>` како директно дете на елементот корен `<web-app>` на следниот начин:

```

1 <servlet>
2   <servlet-name>ime_na_servlet</servlet-name>
3   <servlet-class>ime_na_klasa_bez_ekstenzija</servlet-class>
4 </servlet>

```

Изворен код 2.3: Регистрирање на сервлет во `web.xml`

Во рамките на елементот `<servlet>` можат да се дефинираат и параметри специфични за сервлетот. Овие параметри се дефинираат како име-вредност парови сместени во елементот `<params>`. Детален опис за конфигурација на параметри на сервлет се дадени во додадокот [Б.4](#).

Пресликувањето на шаблонот за URL адресата во соодветен сервлет се дефинира независно од сервлетот во елементот `<servlet-mapping>`. Елементот се сместува на истото хиерархиско ниво како и сервлетот но после неговата дефиниција и се состои од следните елементи:

```

1 <servlet-mapping>
2   <servlet-name>ime_na_servlet</servlet-name>
3   <url-pattern>URL_shablon</url-pattern>
4 </servlet-mapping>

```

Изворен код 2.4: Мапирање на сервлети во `web.xml`

Како вредност на URL шаблон не се вклучува комплетната URL адреса со доменот, туку само релативната адреса во однос на коренот на апликацијата. На пример, ако е потребно да се преслика комплетната адреса "<http://my-spring-e-shop.com/admin/products>" за повик на администрацијската веб апликација (со име `admin`) името на хостот "<http://my-spring-e-shop.com>" и на апликацијата `admin` се занемаруваат и како вредност на URL шаблон се внесува "[/products](#)".

URL шаблонот може да се дефинира и со регуларен израз за мапирање на повеќе URL адреси. На пример, изразот "[/products/*](#)" соодветствува на URL адресите "[/products/e-books](#)", "[/products/software?page=2](#)", но не и на "[/products/software/ide](#)".

Во кодниот сегмент 2.5 е даден пример со дел од `web.xml` датотека во кој се дефинира сервлет за најава `login` кој ќе биде инстанциран од класата `Login.class` и сервлотот за продукти `cart` кој ќе биде инстанциран од класата `ShoppingCart.class`. За барањата чија URL адреса е "[/login](#)" сервлет контејнерот ќе го одбере сервлетот `login`, а за оние со адреса "[/show-cart](#)" ќе го одбере сервлетот `cart`.

```

1 <web-app>
2 ...
3   <servlet>
4     <servlet-name>login</servlet-name>
5     <servlet-class>mk.ukim.finki.Login</servlet-class>
6   </servlet>
7   <servlet>
8     <servlet-name>cart</servlet-name>
9     <servlet-class>mk.ukim.finki.ShoppingCart</servlet-class>
10  </servlet>
11
12 <servlet-mapping>
13   <servlet-name>login</servlet-name>
14   <url-pattern>/login</url-pattern>
15 </servlet-mapping>
16 <servlet-mapping>
17   <servlet-name>cart</servlet-name>
18   <url-pattern>/show-cart</url-pattern>
19 </servlet-mapping>
20 ...
21 </web-app>

```

Изворен код 2.5: Целосна дефиниција на сервлети во дескрипторот за поставување web.xml

Освен во конфигурациската датотека, сервлетите и пресликувањата можат да се дефинираат во самата класа за сервлетот со користење на анотацијата `@WebServlet`. Со ваквиот пристап, во анотацијата се наведува името на сервлетот како вредност на параметарот `name` и мапираната URL адреса како вредност на параметарот `value`. Според тоа, конфигурацијата за сервлетот `cart` од изворниот код 2.5 може да се постигне со анотирање на класата `ShoppingCart` на следниот начин:

```
1 @WebServlet(name = "cart", value = "/show-cart")
2 public class ShoppingCart extends HttpServlet {
3     ...
4 }
```

Користењето на анотацијата `@WebServlet` значително го поедноставува конфигурирањето на сервлетите бидејќи бара многу помалку линии код во дефиницијата на класата за сервлет и ја елиминира потребата за конфигурање во `web.xml`.

Иако анотациите се поедноставни, поинтуитивни и се наоѓаат во самата класа на која се однесуваат, користењето на `web.xml` овозможува веб апликацијата да има една централна датотека во која се наоѓаат сите потребни конфигурации. Неретко програмерите ги употребуваат и двата начини на конфигурација истовремено. Други пак, препорачуваат да се одбере еден тип на конфигурација кон кој ќе се придржуваат за целата веб апликација. Сепак, начинот на конфигурација на сервлетите и на сите останати компоненти кои се дел од една веб апликација е лична префериенца на програмерот.

Во пракса, при дефинирање на сервлет често се употребува анотацијата `@WebServlet` со единствен аргумент кој ја дефинира мапираната URL адреса, без да се наведе името на сервлетот. Со оглед на тоа што името на сервлетот е само за интерна употреба при мапирање, во случаи кога за еден сервлет има само едно мапирање, тоа може слободно да се изостави.

Можна е дефиниција и на повеќе пресликувања на истиот сервлет преку својството `urlPatterns`:

```
1 @WebServlet(urlPatterns = {"/show-cart", "/cart"})
2 public class ShoppingCart extends HttpServlet {
3     ...
4 }
```

Изворен код 2.6: Дефиниција на пресликување со користење на urlPatterns

Поради користењето на регуларни изрази кај мапирањато, можни се ситуации една URL адреса да се пресликува во повеќе сервлети. За разрешување на дилемата кој сервлет да се избере, сервлет контејнерот се води според правила кои се дефинирани во [Б.3.](#)

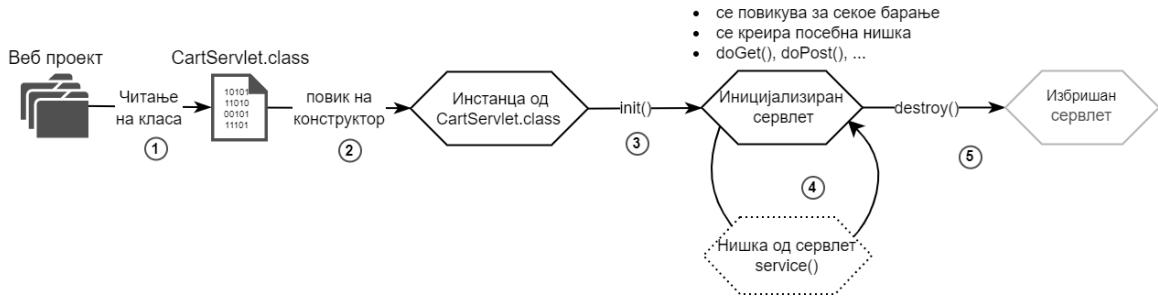
2.3.3 Животен циклус на сервлет

Кога една веб апликација ќе се постави во рамките на веб серверот, сервлетите не се иницијализирани. Одговорноста за управување со комплетниот животен циклус на серверите ја има сервлет контејнерот. За секое пристигнато барање, сервлет контејнерот одлучува кој е одговрен сервлет за негова обработка врз основа на регистрираното мапирање, па пред да го додели барањето на сервлетот, проверува дали тој сервлет е воопшто иницијализиран. Ако станува збор за прво барање за даден сервлет, тогаш контејнерот ја чита класата за негова имплементација, прави инстанца од неа и го повикува методот `init()`. Во текот на целиот животен век, постои само една инстанца од секој сервлет. Во случај да е потребно апликацијата да терминира или да се отстрани од серверот, контејнерот го повикува методот `destroy()` од секој иницијализиран сервлет и ги ослободува сите ресурси кои ги користел.

Дијаграмот на животниот циклус на еден сервлет е прикажан на [слика 2-3](#). Тој се состои од следние чекори кои ги извршува сервлет контејнерот:

1. Ја чита класата во која е имплементиран сервлетот
2. Креира една инстанца од класата
3. Го повикува методот `init()`. Овој метод се повикува само еднаш и во него најчесто се читаат одредени параметри специфични само за сервлетот. После овој метод, сервлетот е подготвен да ги обработува барањата.
4. Го повикува методот `service()` (откако креирал објекти за барање и одговор) како одговор на генерирано барање од клиентот. Методот се извршува во нова нишка.
5. Го повикува методот `destroy()` и животот на сервлетот завршува. Овој метод најчесто се повикува при терминирање на веб апликацијата и се користи за ослободување на ресурси или ажурирање на податоци специфични за сервлетот кои би биле потребни при негова следна иницијализација.

Според дијаграмот, најголем дел од животниот циклус, сервлетот го поминува во состојбата каде го повикува методот `service()` за опслужување на барањата.



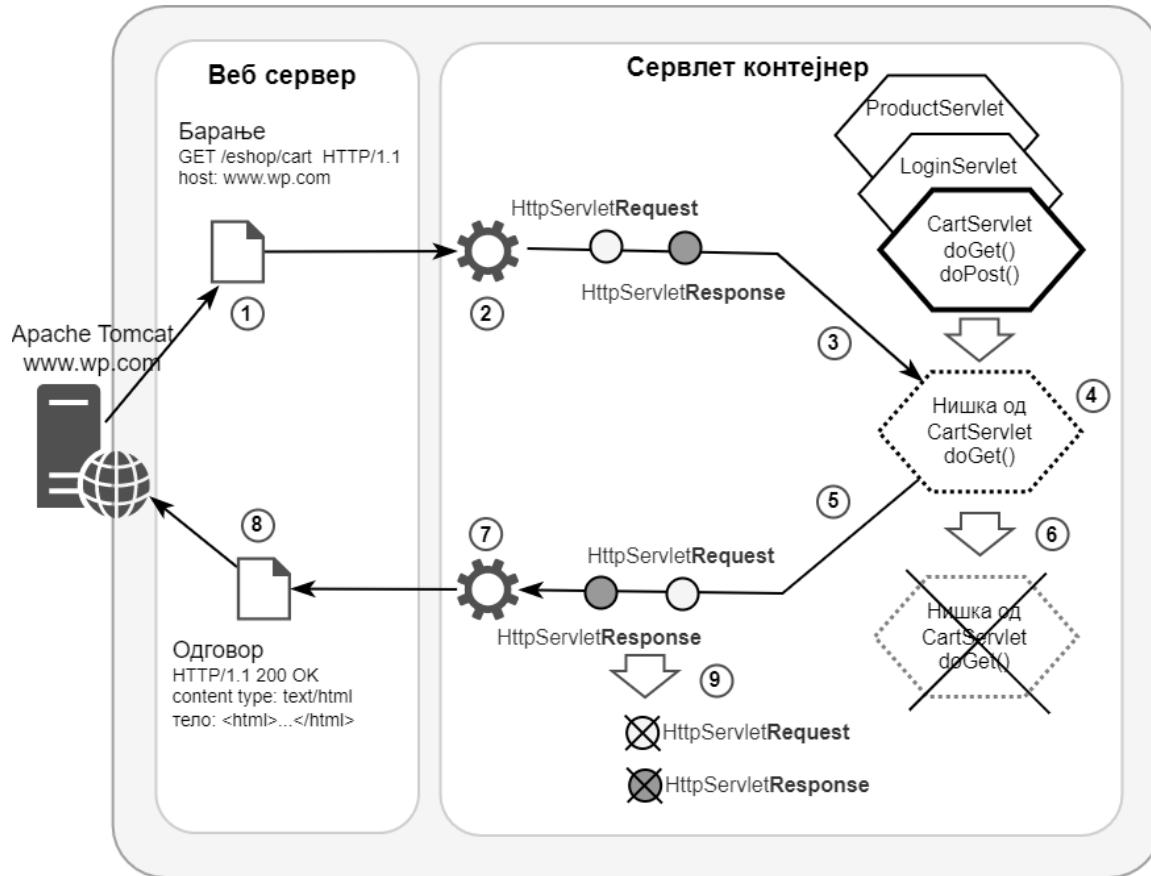
Слика 2-3: Дијаграм на животен циклус на сервлет

Каде предефинираниот начин на инцијализација на сервлети, опслужувањето на првото барање трае подолго време од барањата кои следуваат потоа бидејќи на сервлет контејнерот му е потребно дополнително време да ги обезбеди ресурсите за сервлетот, да ја инстанцира класата и да го изврши методот `init()`. Постои и дополнителен начин на конфигурација каде се задава редоследот на инцијализација на сервлетите веднаш по креирањето на сервлет контејнерот. Овој начин на конфигурација е описан во додадок [Б.5](#).

2.3.4 Обработка на барања од сервлети

Инициализираниот сервлет може да опслужува барање кои му ги делегира сервлет контејнерот. Во продолжение ќе разгледаме подетално како функционира сервлет контејнерот при обработка на дојдовните барања. На сликата [2-4](#) е прикажана постапката на обработка на барање за динамичка веб страница.

1. Веб серверот добива HTTP барање за динамичка страница и му го проследува на сервлет контејнерот
2. Сервлет контејнерот креира објект од класата `HttpServletRequest` која ги содржи сите податоци од HTTP барањето и објект од класата `HttpServletResponse` кој подоцна ќе се наполни со податоци од страна на сервлетот пред да се генерира HTTP одговорот.
3. Сервлет контејнерот одлучува кој веб сервлет да го обработи барањето врз основа на побараната URL адреса и креира нишка од избраниот сервлет. За секое пристигнато барање генерира посебна нишка. Додека се обработува барањето од страна на нишката, сервлетот може да прима и обработува нови барања.
4. Нишката на сервлетот го повикува методот `service()` од сервлетот на кој му се пренесуваат објектите за барање и одговор. Ако се користи `HttpServlet`, што е случај кај веб апликации, `service()` интерно повикува еден од методите за обработка на HTTP барања. Типично, се обработуваат **GET**



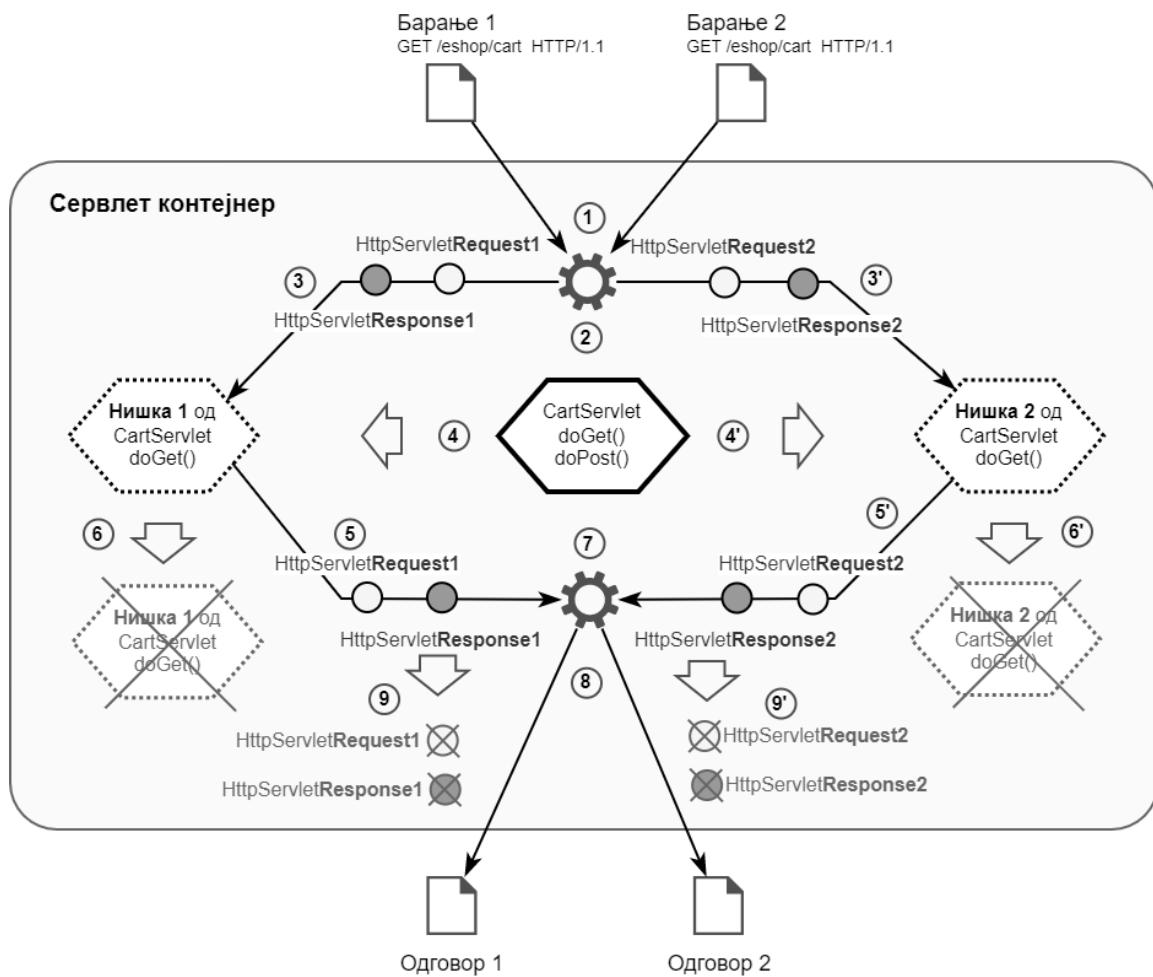
Слика 2-4: Процес на опслужување на барање за динамички ресурс од сервлет контејнер

или **POST** HTTP барања и во тој случај се повикува методот `doGet()` или `doPost()`, соодветно. Овие методи ја имплементираат целата бизнис логика и го полнат објектот за одговор со податоците кои треба да му се испратат на клиентот (динамичката веб страна). Со комплетирањето на овие методи, комплетирана е и задачата на сервлетот.

5. Нишката ги предава објектите на барањето и одговорот назад на сервлет контејнерот.
6. Нишката од сервлетот терминира (се ослободуваат сите ресурси).
7. Контејнерот ги користи податоците од објектот за одговор за да креира HTTP одговор.
8. Контејнерот му го предава HTTP одговорот на веб серверот за негова достава до клиентот.
9. Контејнерот ги брише објектите за барање и одговор и со тоа сите ресурси за обработка на дојдовното барање се ослободени.

Доколку дојдат повеќе паралелни барања за истата динамичка стра-

ница, тогаш сервлет контејнерот генерира посебни инстанци од класите `HttpServletRequest` и `HttpServletResponse` и за секој ваков пар креира посебна нишка од сервлетот кој ќе ги обработи. На крајот, секој сервлет генерира посебен одговор кој се проследува на соодветниот корисник. На слика 2-5 е прикажана постапката на обработка на две паралелни барања од сервлет контејнерот за истиот ресурс. Редоследот на процесите кои се одвиваат во сервлет контејнерот е идентичен како и на слика 2-4 со таа разлика што во овој случај, истите процеси се одвиваат во две паралелни нишки на обработка на секое од барањата.



Слика 2-5: Процес на опслужување на барање за динамички ресурс од сервлет контејнер

2.3.5 Вежба: Имплементација на бизнис логика на сервлет

За да видиме како се имплементира бизнис логиката на сервлетите и да ги видиме некои од начините за интеракција со објектите за HTTP барање и одговор, во

продолжение ќе изработиме вежба во која ќе се испраќаат GET и POST повици до едноставен сервлет преку html страница. За таа цел, креирајте празен веб проект и во именикот "src/main/webapp" креирајте датотека со име index.html и содржина како во кодниот исечок 2.7.

```
1 <html>
2   <body>
3     <a href="servlet">Create empty GET request</a><br>
4     <a href="servlet?q=spring">Create GET request with query string</a>
5     <form action="servlet" method="post">
6       <h2>Insert a value</h2>
7       <input type="text" name="input_data"/>
8       <input type="submit" value="submit">
9     </form>
10   </body>
11 </html>
```

Изворен код 2.7: Содржина на index.html датотека за GET и POST повици до сервлет

Страницата содржи еден линк кој креира празно GET барање, уште еден линк кој креира GET барање во кое се пренесува параметар со име q и вредност spring во низата за пребарување (анг. query string) и копче кое креира POST барање пренесувајќи ги сите вредности од формата simple-form, односно вредноста на текстуалното поле со име input_data. Според вредностите на атрибутите href и action, барањата ќе бидат испратени до URL адресата "/servlet", па за да бидат обработени, потребно е да се креира имплементација на сервлет кој ќе биде пресликан на таа адреса.

Во предефинираниот пакет сместен во именикот "src/main/java", креирајте класа SimpleServlet.class со содржина како во кодниот исечок 2.8. Во кодот е дадена дефиницијата на класата, нејзиното пресликување преку анотацијата @WebServlet, како и имплементација на методите кои се дел од животниот циклус на сервлетот.

```
1 @WebServlet(name = "SimpleServlet", value = "/servlet")
2 public class SimpleServlet extends HttpServlet {
3   public void init() {
4     System.out.println("SimpleServlet is initialized!");
5   }
6
7   public void doGet(HttpServletRequest request, HttpServletResponse
```

```
    response) throws IOException {
8     response.setContentType("text/html");
9     PrintWriter out = response.getWriter();
10    out.println("<html><body>");
11    out.print(getHeaders(request));
12
13    out.println("<h5>" + "Message from doGet() method in SimpleServlet."
14               + "</h5>");
15    if ( request.getParameter("q")!=null)
16      out.println("<h6>" + "Received data:" +
17                  request.getParameter("q") + "</h6>");
18    else
19      out.println("<h5>No data received!</h5>");
20    out.println("</body></html>");
21    int rnd = new Random().nextInt();
22    response.addCookie(new Cookie("firstCookie", String.valueOf(rnd)));
23    System.out.println("SimpleServlet doGet() invoked!");
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
public void doGet(HttpServletRequest request, HttpServletResponse
                     response)
                     throws IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.println("<html><body>");
    out.print(getHeaders(request));
    out.println("<h5>" + "Message from doGet() method in SimpleServlet." +
               "</h5>");
    out.println("<h6>Received data: " +
               request.getParameter("input_data") + "</h6>");
    out.println("</body></html>");
    System.out.println("SimpleServlet doPost() invoked!");
}
public void destroy() {
    System.out.println("SimpleServlet is destroyed!");
}
private String getHeaders(HttpServletRequest request){
    StringBuilder builder = new StringBuilder();
```

```

43     Enumeration<String> headerNames = request.getHeaderNames();
44     while(headerNames.hasMoreElements())
45     {
46         String header = headerNames.nextElement();
47         builder.append(header+": "+request.getHeader(header)+"<br> ");
48     }
49     return builder.toString();
50 }
51 }
```

Изворен код 2.8: Имплементација на сервлет servlet во класата SimpleServlet.class

Методите init() и destroy() се повикуваат само еднаш: кога за прв пат ќе се прочита класата и кога таа ќе се избрише при терминирање на апликацијата. Имплементацијата на овие методи содржи код за печатење на текстуална порака на конзолата во IDE чија намена е само да го означи моментот на нивното извршување. Класата ги содржи и методите doGet() и doPost(). И двата методи го поставуваат типот на одговорот да биде "text/html" со цел прелистувачот да го третира како содржина на веб страница која треба да ја прикаже. Дополнително, методите испишуваат порака на конзолата на IDE кога ќе бидат повикани. И двата методи го повикуваат приватниот метод getHeaders() кој ги изминува сите заглавја од барањето и на крајот враќа текстуална низа со нивните имиња и вредности. Оваа низа се вметнува во одговорот за на клиентот да му се прикажат сите заглавја кои неговиот прелистувач ги испратил до севлетите.

Методот doGet() проверува дали во низата за пребарување прелистувачот испратил параметар со име q. Ако постои таков параметар, го печати. Во спротивно, враќа порака дека не примил никакви податоци. На крајот, методот додава колаче со име firstCookie чија вредност е случајно генериран цел број. Методот doPost() има слична улога, но тој ја чита содржината на параметарот со име input_data од телото на барањето и во одговорот ја запишува содржината на сите заглавја и вредноста на параметарот.

Ако ја извршиме апликацијата и ја повикаме страницата index.html, во прелистувачњот се добива преглед со следната содржина:

[Create empty GET request](#)
[Create GET request with query string](#)

Insert a value

Servlet	submit
---------	--------

Ако кликнеме на првиот линк, ќе ја добијеме следната порака во прелистувачот:

```
host: localhost:8080
connection: keep-alive
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="102", "Google
Chrome";v="102"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
upgrade-insecure-requests: 1
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.0.0
Safari/537.36
accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/web
exchange;v=b3;q=0.9
sec-fetch-site: same-origin
sec-fetch-mode: navigate
sec-fetch-user: ?1
sec-fetch-dest: document
referer: http://localhost:8080/ch2_handson_1_war_exploded/
accept-encoding: gzip, deflate, br
accept-language: en-GB,en;q=0.9
```

Message from doGet() method in SimpleServlet.

No data received!

Бидејќи за секој повикан метод додадовме код за испишување на порака на конзола, во конзолата на IDE ќе бидат испишани следните пораки:

-
- 1 SimpleServlet **is initialized!**
 - 2 SimpleServlet **doGet() invoked!**
-

Можеме да забележиме дека со првиот повик за сервлетот се повикува методот init(), а потоа, откако сервлетот ќе биде иницијализиран, се повикува методот doGet() за обработка на GET барањето испратено од линкот од страницата.

Ако се вратиме назад и кликнеме на вториот линк, ќе ја добијеме следната порака во прелистувачот (на слика е прикажан парцијален изглед од кој се изоставени поголем дел од заглавјата): Можеме да забележиме дека, за разлика

```
cookie: firsCookie=-1193959466
```

Message from doGet() method in SimpleServlet.

Received data:spring

од првиот повик, кај овој повик веќе има колаче кое беше додадено при првото барање. Ова колаче ќе биде присутно во секој нареден одговор.

На конзолата ќе биде допишана пораката: `SimpleServlet doGet() invoked!`

Бидејќи сервертот е веќе иницијализиран при обработката на првото барање, методот `init()` нема повторно да се повика.

Ако ја внесеме вредноста `Servlets` во текстуалното поле и ако притиснеме на копчето, ќе биде испишана пораката:

```
host: localhost:8080
connection: keep-alive
content-length: 20
cache-control: max-age=0
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="102", "Google
Chrome";v="102"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
upgrade-insecure-requests: 1
origin: http://localhost:8080
content-type: application/x-www-form-urlencoded
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.0.0
Safari/537.36
accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/we
exchange;v=b3;q=0.9
sec-fetch-site: same-origin
sec-fetch-mode: navigate
sec-fetch-user: ?1
sec-fetch-dest: document
referer: http://localhost:8080/ch2_handson_1_war_exploded/
accept-encoding: gzip, deflate, br
accept-language: en-GB,en;q=0.9
cookie: firsCookie=-1193959466
```

Message from doPost in SimpleSerlvet.

Received data: post-data

Бидејќи се испраќаат податоци во телото на барањето, во барањето се пренесуваат и заглавјата `content-length` и `content-type`.

На конзолата ќе биде испишана пораката `SimpleServlet doPost() invoked!`

За крај, ако го прекинеме извршувањето на апликацијата во IDE, сервертот ќе го повика методот `destroy()`, па затоа и на конзолта ќе биде испишана пораката `SimpleServlet is destroyed!`

2.3.6 Интеракција помеѓу серверти

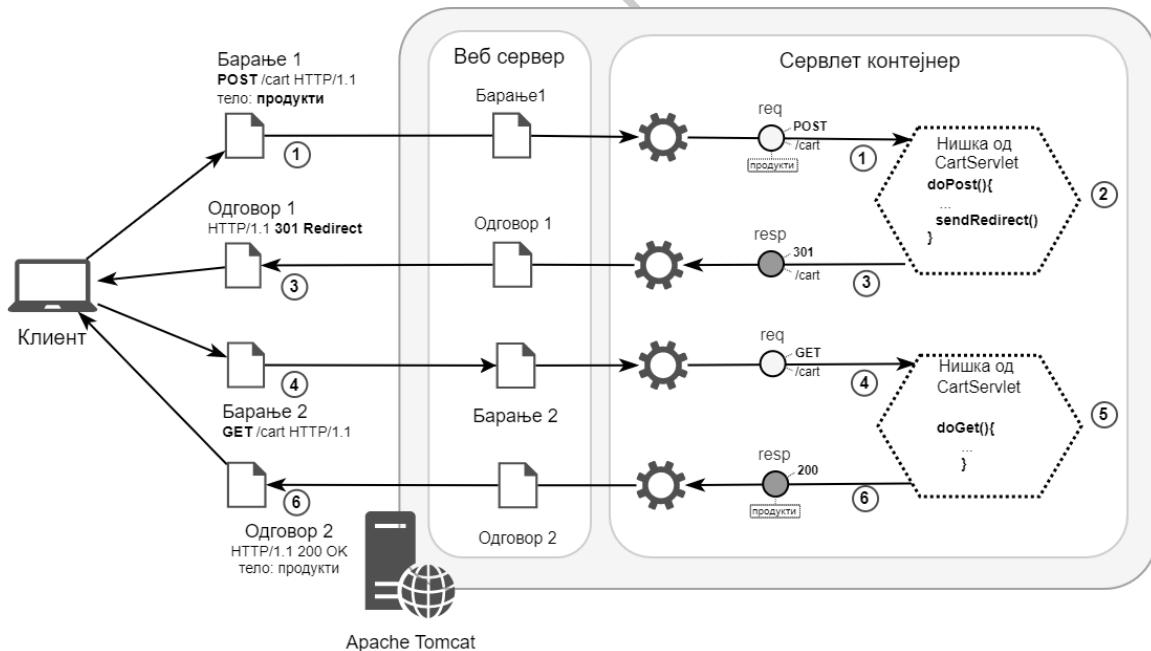
Во одредени ситуации, сервертите не можат делумно или во целост да го обработат пристигнатото барање, па затоа тие го проследуваат до други серверти

или ресурси преку интерфејсите за интеракција. Наједноставниот начин да се препрати барање кон друг ресурс, без разлика дали станува збор за ресурс од самата апликација или надворешна URL адреса, е преку методот `sendRedirect` од интерфејсот `HttpServletResponse` кој го има следниот потпис:

```
public void sendRedirect(String URL) throws IOException;
```

Како аргумент на методот се пренесува URL адресата кон која треба да се пренасочи барањето. Методот ја поставува оваа адреса како вредност на полето `Location` на HTTP одговорат и истовремено го поставува неговиот статус на вредност 301 (Redirect). Откако прелистувачот на клиентот ќе го добие ваквиот одговор, веднаш го препраќа корисникот кон новата адреса наведена во `Location`. Важно е да се напомене дека пренасочувањето се одвива на клиентска страна бидејќи прелистувачот генерира новото барање.

Во исечокот 2.9 е даден пример за сервлетот `CartServlet` имплементиран во класата `ShoppingCartServlet` пресликан во URL адресата `/cart` со `doGet` метод кој враќа листа на продукти од потрошувачката кошничка на корисникот (го идентификува според податоците од колачето во барањето) и `doPost` метод чија улога е да прочита податоци за продукти кои се пренесуваат во телото на барањето и да ги внесе во базата на податоци како продукти кои се дел од кошничката на корисникот. Комуникацијата помеѓу клиентот и веб апликацијата од примерот е илустриран на слика 2-6.



Слика 2-6: Дијаграм на животен циклус на сервлет

1. Корисникот додал нови продукти и притиснал на копче со кое испратил

POST барање до `"/cart"` во чие тело се наоѓаат податоците за новите продукти.

2. Барањето пристигнува до сервлетот кој во методот `doPost()` ги додава новите податоци во базата. Прелистувачот на корисникот сèуште ја прикажува старата содржина на кошничката, па за да се прикаже ажурираната листа на продукти во кошничката, потребно е да се натера прелистувачот да направи GET барање со кое би се побарале податоци за продуктите преземени од серверот. Затоа, на крајот од методот `doPost()` се повикува методот `sendRedirect()` кој генерира одговор со статус 301 (Redirect)
3. Одговорот се испраќа до клиентот
4. Како резултат на одговорот со статус 301, прелистувачот веднаш генерира ново GET барање кон адресата `/cart`. Клиентот не презема никава акција и не е свесен за новото испратено барање.
5. Барањето пристигнува до сервлетот кој во методот `doGet()`, во нова и различна нишка од претходната, ги презема сите продукти од кошничката за корисникот и генерира одговор
6. Серверот го испраќа одговорот до клиентот чиј прелистувач ја прикажува новата листа на продукти. URL адресата во прелистувачот се менува со URL адресата на новиот повик.

Во случај корисникот да ја освежи страницата (анг. refresh), ќе биде испратено само GET барањето, но не и POST барањето. Оваа карактеристика е од голема важност кога се испраќа барање за менување на податоци во базата како што се бришење (DELETE), додавање (POST) и уредување (PUT), бидејќи независно колку пати се освежува страницата, промената ќе биде направена само еднаш. Токму тоа е и главната примена на `sendRedirect()`. Друга примена на овој методот е во случаите кога потребно барањето да биде обработено од надворешен ресурс.

```

1  @WebServlet(name = "cartServlet", value = "/cart")
2  public class ShoppingCartServlet extends HttpServlet {
3      protected void doGet(HttpServletRequest request, HttpServletResponse
4          response) throws IOException {
5              // fill the response with html representing shopping cart
6      }
7      protected void doPost(HttpServletRequest request, HttpServletResponse
8          response) throws IOException {
9          // retrieve data from request
10         // update data into database
11         response.sendRedirect("cart");
12     }
13 }
```

```

10    }
11 }
```

Изворен код 2.9: Пример за редирекција

Со помош на `sendRedirect` клиентот може да се пренасочи и на надворешни адреси на домени кои се различни од доменот на апликацијата. Сепак, ваквото пренасочување има и свои недостатоци. Еден од нив е дополнително чекање поради испраќањто на одговорот за редирекција назад до клиентот, за потоа да се испрати барање до новата локација и да се добие конечен одговор од таа локација. Покрај доцнењето, се воведува и дополнителен мрежен сообраќај. Друг голем недостаток на `sendRedirect()` е што се губи целата содржина на препратеното барањето, па било која информација која ја испратил клиентот во телото е недостапна за дестинацискиот ресурс.

За да се надминат недостатоците на препраќањето се користи интерфејсот `RequestDispatcher` кој ги нуди методите `forward()` и `include()` за проследување на барањето до друг ресурс во рамките на апликацијата.

```

1 public void forward(ServletRequest request,ServletResponse response) throws
   ServletException,java.io.IOException
2 public void include(ServletRequest request,ServletResponse response) throws
   ServletException,java.io.IOException
```

Инстанца од распоредувачот на барања `RequestDispatcher` се добива ако се повика методот `getRequestDispatcher` од објектот на барањето (`HttpServletRequest`) во рамките на сервлетот. Потписот на методот е: `public RequestDispatcher getRequestDispatcher(String resource)`; Како аргумент на методот се пренесува релативна адреса на сервлетот или ресурсот на кој треба да се проследи барањето. Октако ќе се повика методот `forward()`, барањето се предава на новиот сервлет. Доколку изворниот сервлетот кој го повикува методот ја променил содржината на барањето, тоа се пренесува до дестинацискиот сервлет со променета содржина. Изворниот сервлет може да ја менува содржина на одговорот до моментот на повикување на методот `forward()`, но после повикот, потокот за запишување во телото на одговорот се затвора и не може да менува во рамките на изворниот сервлет. Дестинацискиот сервлет продолжува со процесирање на барањето, и откако ќе заврши со уредување на одговорот, го враќа одговорот директно на клиентот. Линиите на код кои следат по `forward()` се извршуваат, но немаат никакво влијание врз одговорот бидејќи самото повикување на `forward()` го затвора запишувањето во него. Изворниот сервлет повеќе не може на ниту еден начин да влијае врз одговорот, ниту пак да го испрати одговорот назад до клиентот. Со ваквиот начин на обработка, целата контрола се префрла на дестинацискиот сервлет. Главната примена на методот `forward()` е во случаите

кога барањето треба да биде дополнително обработено од друг сервлет или пак во Модел-Преглед-Контролер концептот каде барањето се предава на прегледот како ресурс (jsp, thymeleaf инт) со цел да се генерира конечната содржина која ќе се испрти до клиентот.

Ако се навратиме на примерот за сайт за електронска продавница, употребата на методот `forward()` би била од голема полза кај сервлетот кој врши автентификација на корисник. Во кодниот исечок 2.10 е даден пример на код за сервлет за автентификација `login` кој го обработува POST барањето за автентификација. Сервлетот ги зема корисничкото име и лозинка од телото на барањето и, доколку сè е во ред, истото (или модифицирано) барање го препраќа на дестинациски сервлет за потрошувачка кошничка со URL адреса `"/cart"`, кој враќа одговор до корисникот со листа на продукти во кошничката. Дестинацискиот сервлет може да го идентификува корисникот според податоците во оригиналното барање или пак дополнителните податоци кои изворниот сервлет ги вnel пред да го проследи. Како метод кој ја презема контролата врз извршувањето, `forward()` испраќа одговор до корисникот.

```

1  @WebServlet(name = "login", value = "/login")
2  public class LoginServlet extends HttpServlet {
3      protected void doPost(HttpServletRequest request, HttpServletResponse
4          response) throws IOException, ServletException {
5          // retrieve username and password from request
6          String username = request.getParameter("username");
7          String password = request.getParameter("password");
8          if (username.equals("admin") && password.equals("!@34")){
9              RequestDispatcher dispatcher =
10                 request.getRequestDispatcher("cart");
11                 dispatcher.forward(request, response);
12             }
13             else {
14                 response.setContentType("text/html");
15                 PrintWriter out = response.getWriter();
16                 out.print("Invalid user name or password!");
17                 RequestDispatcher
18                     dispatcher=request.getRequestDispatcher("/login.html");
19                     dispatcher.include(request, response);
20             }
21         }
22     }

```

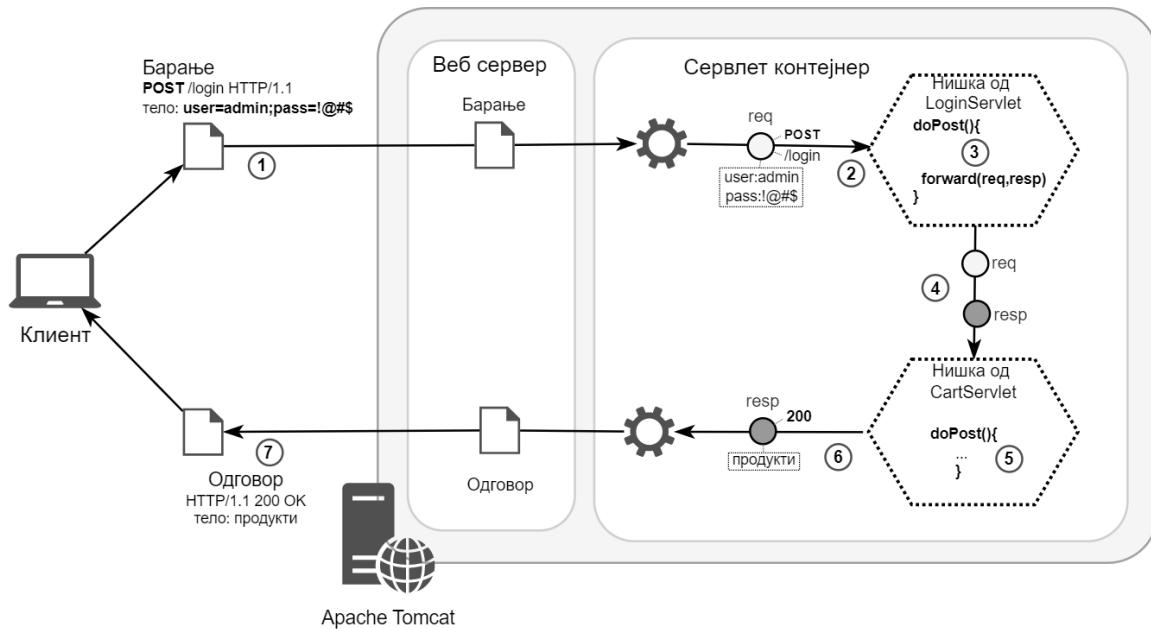
Изворен код 2.10: Пример за `forward()` и `include()`

Сценариото од комуникацијата помеѓу клиентот и веб апликацијата е прикажано на слика 2-7 и се одвива во следните чекори:

1. Клиентот внесува корисничко име и лозинка и ги испраќа како POST барање до веб серверот
2. Објектот за барањето пристигнува до сервлетот `/login`, (инстанца од `LoginServlet`)
3. Сервлетот го обработува барањето во методот `doPost()`, ги чита податоците кои се пренесуваат во неговото тело и откако ќе го верификува корисникот, го повикува методот `forward()` за да се проследи барањето до сервлетот `cart` (инстанца од `CartServlet`).
4. `RequestDispatcher` го проследува барањето до новиот ресурс, задржувајќи го типот на метод (POST) и сите останати параметри. Иако во нормални случаи одговорот на излез од сервлетот се враќа до клиентот, при повик на `forward()` одговорот, заедно со барањето се проследуваат до следниот ресурс.
5. Сервлетот `/cart` го обработува барањето во методот `doPost()`. Метод ги чита продуктите за корисникот од база и ги внесува во одговорот, и потоа го предава одговорот на контејнерот.
6. Контејнерот креира HTTP одговор, го испраќа кон веб серверот и ги брише објектите за барањето и одговорот
7. Контролата врз извршувањето се враќа во првата наредба после `forward()` во изворниот сервлет `/login`, но бидејќи објектот за запишување е веќе затворен, не може да се прават промени врз него (обично нема код во овој дел)⁷
8. Прелистувачот на клиентот ја прикажува листата на продукти за најавениот корисник. URL адресата на прелистувачот останува непроменета (за повик на сервлетот `/login`), иако одговорот е добиен од сервлетот `/cart`.

За разлика од `forward()`, кај методот `include()`, контролата останува кај изворниот сервлет. Неговата главна примена е во случаите кога е потребно во телото на одговорот се вклучи дополнителен ресурс. Ако дестинацискиот ресурс е динамички, на пример сервлет, откако ќе го повика методот `include()`, изворниот сервлет го препраќа барањето и одговорот до дестинацискиот ресурс кој може да го менува објектот на одговорот и потоа повторно го враќа назад кај изворниот сервлет. Динамичкиот дестинациски ресурс има ограничени привилегии во менување на одговорот, односно, може да го менува само неговото тело (потокот

⁷Доколку има потреба од промени во иницијалниот сервлет или филтер, се користи `ResponseWrapper`, но овој дел нема да биде објаснет во оваа книга, бидејќи ја надминува предвидената комплексност.



Слика 2-7: Метод forward()

за запишување во телото на одговорот останува отворен), но не и вредностите на останатите полиња од HTTP заглавјето, вклучувајќи го и статусот на одговорот. Откако изворниот сервлет ќе го добие променетиот одговор, може да продолжи со негово уредување пред да го испрати до корисникот.

Ако станува збор за статички дестинациски ресурс, на пример html страна, тогаш сè што било запишано во одговорот од страна на изворниот сервлет пред да се повика `include()` се вклучува во одговорот, кој освен статичкиот ресурс, во телото ќе ги содржи и податоците испишани од страна на изворниот сервлет. Дополнително, во телото може да се вклучат и нови податоци во наредбите што следат опсле повикот на `include()` во изворниот сервлет.

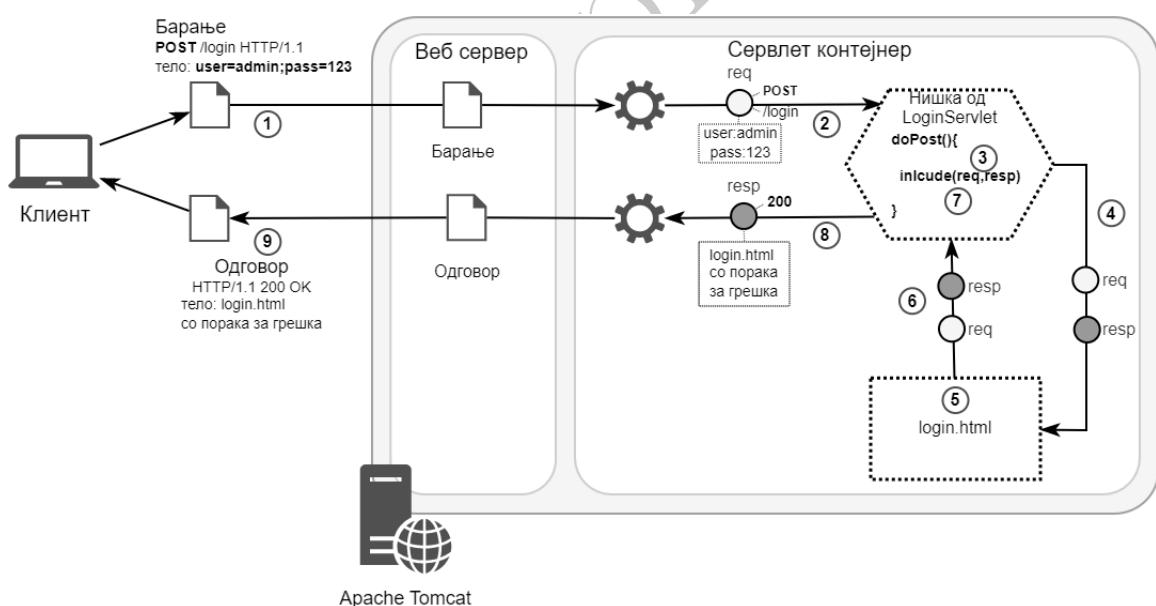
Методот `include()` најчесто се користи кога е потребно да се вклучи дополнителна содржина од типот на банер, заглавје, подножје во рамките на ресурсот (html, jsp и слично) кој се испраќа како одговор.

Употребата на `include()` е прикажана во примерот со автентификација на корисник кај сајтот за електронска продавница во кодниот исечок 2.10. Чекорите од целокуната комуникација се илустрирани на слика 2-8:

1. Клиентот внесува корисничко име и потрешна лозинка кои ги испраќа како POST барање до веб серверот
2. Објектот за барањето пристигнува до сервлетот `login`, (инстанца од `LoginServlet`)
3. Сервлетот го обработува барањето во методот `doPost()`, ги чита податоците кои се пренесуваат во неговото тело и откако не успеава да го верифику-

ва корисникот поради погрешна лозинка, испишува порака за неуспешна автентикација во одговорот, а потоа го повикува методот `include()` за да вклучи во одговорот статички ресурс кој ја содржи страницата за најава `login.html`.

4. Барањето и одговорот се предаваат на `RequestDispatcher`
5. `RequestDispatcher` го вклучува статичкиот ресурс во одговорот или, во случај на сервлет, го повикува новиот сервлет
6. Откако статичкиот ресурс е вклучен, или во случај на динамички ресурс (сервлет), откако ќе биде обработено барањето и одговорот, истите се пролесдуваат назад до изворниот сервлет `/login`. Одговорот сеуште не се испраќа на корисникот.
7. Сервлетот `/login` го добива назад одговорот и контролата врз извршувањето, ги извршува линиите код кои следат после `include()` и го предава објектот на одговорот на контејнерот
8. Контејнерот креира HTTP одговор, го испраќа кон веб серверот и ги брише објектите за барањето и одговорот
9. Прелистувачот на клиентот ја прикажува формата за најава од статичкиот ресурс. URL адресата на прелистувачот останува непроменета



Слика 2-8: Метод `include()`

Како генерелен заклучок, главната разлика помеѓу овие два метода е што `forward()` и го затвора одговорот за запишување и ја предава контролата на дестинацискиот ресурс кој потоа го испраќа одговорот до клиентот, додека `include()` го предава одговорот за да се вклучи во него дополнителен ресурс, задржувајќи

ја целосно контролата врз одговорот, па затоа и одговорот се враќа назад до клиентот од сервлетот кој го повикал методот.

2.4 Управување со состојби

2.4.1 Параметри и атрибути

Во рамките на една веб апликација потребно е да се чуваат перманентни или привремени податоци во различен облик со различен опсег на важење. Кај Јава серверското издание овие податоци можат да бидат зачувани во облик на **параметри** или **атрибути**. Параметрите се карактеризираат по тоа што можат да чуваат само текстуална вредност која не може да се менува еднаш штом се постави. Пример за параметри се податоците што се испраќаат од клиентот како дел од низата за пребарување во барањето. Овие податоци се поставени од клиентите, и се испратени како низа од знаци, па затоа, при обработката на барањето од страна на сервлетот, тие може само да се прочитаат. Параметрите може да се користат и за складирање на податоци како дел од дескрипторот за поставување кои се читаат при инцијализација на сервлет контејнерот или на некој сервлет. За читање на параметар се користи методот `String getInitParameter(String ime)` кој ја враќа текстуалната вредност на параметарот со името кое се пренесува како аргумент. Ако се чита целоброжна вредност на параметар, тогаш таа повторно се третира како податок од типот `String`.

За разлика од параметрите, атрибутите се податоци кои може да се уредуваат една штом се постават. Нивниот тип не е ограничен само на текст, туку можат да бидат било кој тип кој може да се серијализира. Тие не се поставуваат Дескрипторот на поставување, туку во самиот код на менацираните компонети. Како и параметрите, имаат различен опсег на важење. На пример, постојат контекстни атрибути до кои може да се пристапи од било која менацирана компонента, или пак атрибути кои се поставуваат како дел од барање и можат да се уредуваат само од оние компоненти на кои им се проследува објектот на барањето. Атрибутите се читаат преку методот `Object getAttribute(String ime)` кој ја враќа вредноста на атрибутот со името кое се пренесува како аргумент. Оваа вредност мора експлицитно да се конвертира (cast) во типот на објект кој се пренесува, во спротивно, ќе се исфрли грешка. Вредноста на атрибутите може да се менува и за таа цел се користи методот `setAttribute(String ime, Object vrednost)` кој на атрибутот со даденото име му ја доделува дадената вредност.

Според тоа од каде може да им се пристапи на параметрите и атрибутите во рамките на една веб апликација се дефинираат следните области на важење:

- Сервлет - важи само за параметри и тоа на ниво на сервлет
- Контекст - важи и за параметри и за атрибути и на ниво на цела апликација
- Барање - важи и за параметри и за атрибути на ниво на барање
- Сесија - важи само за атрибути на ниво на цела апликација, но само за еден ист корисник

Комплетен преглед на разликите помеѓу параметрите и атрибутите е даден на слика 2-9.

	Атрибут	Параметар
Опсег на важност	ServletContext Request Session	ServletContext ServletConfig Request
Тип на податок	Object	String
Метод за читање	getAttribute(String ime)	getInitParameter(String ime)
Метод за поставување	setAttribute(String ime, Object vrednost)	Не може да се постави во код, туку во web.xml или преку анотации

Слика 2-9: Преглед на карактеристики на атрибути и параметри

2.4.2 Работа со параметри и атрибути во апликациски опсег (ServletContext)

При иницијализацијата на апликацијата (пред да се иницијализираат сервлетите), сервлет контејнерот креира контекст, односно, инстанца од класата `ServletContext` која преставува објект во кој се сместуваат референци кон сите конфигурациски параметри за апликацијата поставени во конфигурациската датотека `web.config`, но и кон кориснички атрибути кои можат да се додаваат и уредуваат во текот на животниот век на апликацијата. Контекстот претставува централен објект до кој можат да пристапуваат сите сервлети и на тој начин овозможува споделување на параметри и атрибути на ниво на целата апликација.

Пристап до ServletContext

Објектот `ServletContext` е од големо значење за апликацијата, па затоа постојат неколку начини да се пристапи до него и неговите податоци. Веднаш по креирањето на `ServletContext`, сервлет контејнерот ја зачува неговата референца и при секоја иницијализација на сервлет ја запишува истата референца во објектот `ServletConfig` кој се креира за самиот сервлет. Според тоа, едниот начин на

пристан до `ServletContext` е преку класата `GenericServlet` (од која се изведува `HttpServlet` која го нуди методот `public ServletContext getServletContext()`

Друг начин на пристап до `ServletContext` е преку интерфејсот `ServletConfig` кој го нуди истиот метод `getServletContext()`. Во кодниот исечок 2.11 во продолжение, прикажани се двата начини на пристап до `ServletContext` во рамките на сервлетот `login`.

```

1 @WebServlet("/login")
2 public class LoginServlet extends HttpServlet {
3     public void init() {
4         ServletContext context1 = getServletContext();
5         ServletConfig config = getServletConfig();
6         ServletContext context2 = config.getServletContext();
7     }
8 }
```

Изворен код 2.11: Пример за пристап до референца од `ServletContext`

Дефиниција и пристап до контекстни параметри

Апликациските параметри, или поточно, контекстуалните параметри се дефинираат во `web.config` на сличен начин како и параметрите на сервлетите, со таа разлика што, наместо во елементите за сервлети, тие се сместени помеѓу ознаките `<context-param>...</context-param>` како први деца на коренот `<web-app>` на xml дрвото во `web.config`.

Пример за конфигурација на контекстни параметри е даден во кодниот исечок 2.12.

```

1 <web-app>
2 ...
3     <context-param>
4         <param-name>admin-user</param-name>
5         <param-value>administrator</param-value>
6     </context-param>
7     <context-param>
8         <param-name>admin-password</param-name>
9         <param-value>5pr1ng</param-value>
10    </context-param>
11 ...
12 </web-app>
```

Изворен код 2.12: Дефиниција на контекстни параметри во web.xml

Методите за пристап до контекстните параметри се идентични како и кај интерфејсот `ServletConfig`, односно, може да се пристапи до поединечен параметар преку неговото име или да се добие енумерација со имињата на сите параметри кои можат да се искористат за пристап до нивните вредности. Методите за пристап до контекстни параметри на интерфејсот `ServletContext` се

```
1 public String getInitParameter(String name)  
2 public Enumeration<String> getInitParameterNames()
```

Управување со контекстни атрибути

Освен споделување на параметри од конфигурациската датотека, `ServletContext` овозможува споделување на атрибути креирани во кодот од страна на корисниците. Како атрибут може да биде било кој тип на податок или објект. Атрибутите можат да бидат креирани, уредувани и бришени и од страна на сервлет контејнерот и од страна на апликацискиот код во сервлетите. Процесот на додавање на атрибут во контекстот се вика поврзување (анг. binding) бидејќи во контекстот се додава референца кон објектот преку која можат да пристапат сите корисници кои имаат пристап до контекстот. Методите на интерфејсот `ServletContext` за управување со атрибути се:

```
1 public void setAttribute(String name, Object object)  
2 public Object getAttribute(String name)  
3 public void removeAttribute(String name):
```

Првиот метод се повикува со име кое се доделува на атрибутот и негова вредност која е од генеричкиот тип `Object` и може да биде било каков објект. Ако веќе постои атрибут со тоа име во контекстот, тогаш неговата вредност се ажурира со новата вредност. Ако таков атрибут не постои, тогаш тој се додава во контекстот. Името на атрибутот се користи за да се пристапи до неговата референца преку методот `getAttribute()` за читање, уредување или пак за негово отстранување од контекстот со помош на методот `removeAttribute()`. Освен со последниот метод, атрибут може да се отстрани од контекстот ако му се постави вредност `null`.

Во кодниот исечок 2.13 е даден пример за управување со контекстен атрибут од типот `Integer` со име `cartVisits` на сервлетот `cart`. При самата иницијализација на сервлетот се поставува вредноста на атрибутот на 0. Ова не е типично сценарио за иницијализација на контекстни атрибути, но ни служи само за илустрација (подоцна ќе биде разгледан настанот кој се користи за иницијализација). Потоа, за секое GET барање се чита вредноста на атрибутот, се инкрементира за

1 и повторно се запишува во контекстот. Важно е да се забележи дека при читање на атрибутите од контекстот мора да се направи претворање во соодветен тип на податок (анг. cast) бидејќи методот `getAttribute` секогаш враќа инстанца од најопштата класа `Object` од која се изведени сите класи, па затоа нема начин како да се знае кој е конкретниот тип на податок на атрибутот. Токму поради овој факт, вредноста на даден атрибут може да се препокрие со вредност од различен тип без да настане грешка во извршувањето. При работа со атрибути, програмерите треба да водат контрола за типот на податок на атрибутот при неговото првично додавање во контекстот за да можат да прават соодветно претворање при секое читање на тој атрибут низ кодот. На крајот од примерот, во методот `destroy()` се прави отстранување на атрибутот од контекстот.

Важно е да се забележи дека до атрибутот може да се пристапи и да се менува неговата вредност од страна на било кој сервлет, односно, неговата употреба не е ограничена на конкретниот сервлет во кој е дефиниран.

```

1 @WebServlet("/cart")
2 public class ShoppingCartServlet extends HttpServlet {
3     public void init() throws ServletException {
4         super.init();
5         Integer visits = 0;
6         getServletContext().setAttribute("cartVisits", visits);
7     }
8     public void doGet(HttpServletRequest request, HttpServletResponse
9                         response) throws IOException {
10        ServletContext context = getServletContext();
11        Integer visits = (Integer) context.getAttribute("cartVisits");
12        getServletContext().setAttribute("cartVisits", ++visits);
13    }
14    public void destroy() {
15        super.destroy();
16        getServletContext().removeAttribute("cartVisits");
17    }
}

```

Изворен код 2.13: Пример за уредување на контекстни атрибути

Преглед на настани, параметри и атрибути во контекстен домен

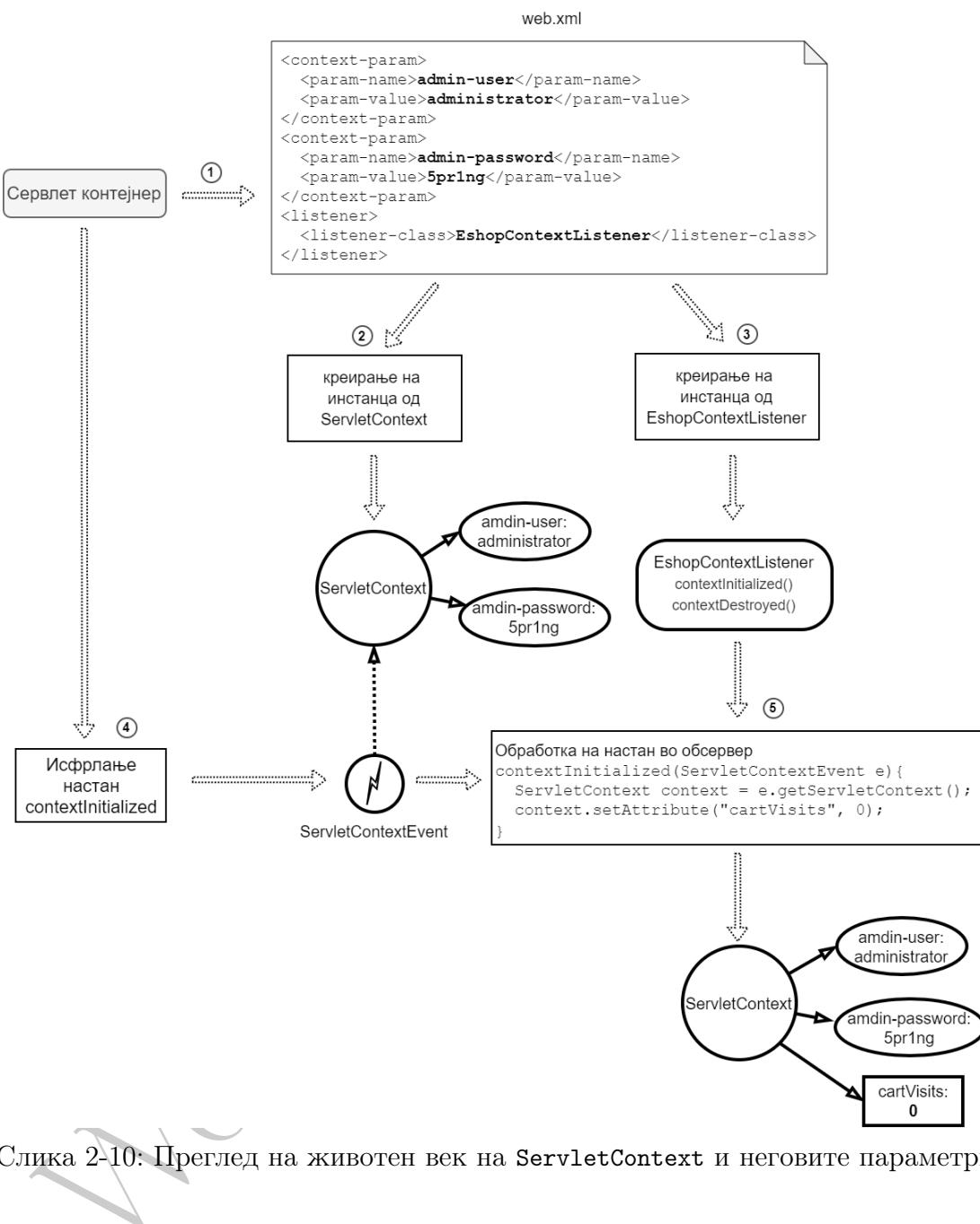
При користење на контекстни атрибути, често е потребно тие да се иницијализираат со почетни вредности и истите да се избришат кога апликацијата ќе заврши. Поради нивниот глобален опсег на важење на ниво на целата апликација, важно е вредностите да се иницијализираат пред да бидат креирани сервлетите и да

бидат избришани откако ќе се избришат сите сервлети. За да се доловат овие моменти, секогаш кога има промени во состојбата на контекстот, а тоа се негово креирање или бришање, сервлет контејнерот еmitува настан кој е инстанца од типот `SerlvetContentEvent`. Настанот содржи референца кон `SerlvetContentxt` за да овозможи пристап до контекстот при негова обработка. Настанот се обработува од страна на обсервер кој преставува менацирана класа што го имплементира интерфејсот `ServletContextListener` со методи за обработка на настаните испалени при иницијализација и терминирање на контекстот. Како менацирана класа, обсерверот треба да се регистрира со цел да може да биде препознаен и иницијализиран од сервлет контејнерот. Подетален опис на настаните поврзани со контекстот и нивна обработка се дадени во додадок [Б.6](#).

На слика [2-10](#) е даден животниот циклус на контекстот, настаните поврзани со него како и неговата врска со контекстните параметри и атрибути.

1. При вклучување на веб апликацијата (отпакување на `.war`), откако серверот ќе креира го сервлет контејнерот, сервлет контејнерот започнува со читање на дескрипторот за поставување `web.xml`
2. Сервлет контејнерот креира инстанца од `ServletContext` и во неа поставува референции кон контекстните параметри поставени во `web.xml`
3. Сервлет контејнерот продолжува со читање на `web.xml` и скенирање на менацираните компоненти. Откако ќе го прочита регистрираниот обсервер на контекстни настани кој го имплементира интерфејсот `ServletContextListener`, креира негова инстанца
4. `ServletContext` е веќе иницијализиран, па затоа сервлет контејнерот исфрла настан за иницијализација преку објектот `ServletContextEvent`. Сервлет контејнерот му додава референца на настанот кон `ServletContext` со цел да може да се пристапи до контекстот при неговата обработка
5. Инстанцата на обсерверот на контекстни настани го фаќа настанот и го обработува. Бидејќи станува збор за промена на состојба на контекстот, односно контекстот е иницијализиран, се извршува кодот на методот `contextInitialized(ServletContextEvent e)` во кој како аргумент се пренесува инстанца од објектот кој го презентира настанот за промена на контекст. Овој објект содржи референца кон `ServletContext`, па затоа преку него може да се пристапи до контекстот и во него да се постават иницијални вредности на контекстните атрибути.

На крајот од овој процес, контекстот содржи референции кон контекстните параметри и контекстните атрибути до кои може да се пристапи од било која менацирана компонента во текот на целиот животен век на веб апликацијата.

Слика 2-10: Преглед на животен век на `ServletContext` и неговите параметри

Иако не е прикажано на сликата, откако ќе се затвори апликацијата, пред да се избрише `ServletContext`, сервлет контејнерот го исфрлува настанот `contextDestroyed` преку инстанца од `ServletContextEvent` која ќе биде пренесена како аргумент на методот `contextDestroyed(ServletContextEvent e)` во обсерверот. Овој настан може да се искористи за зачувување на одредена вредност на контекстните атрибути во некое перманентно податочно складиште со цел да се реискористи при следното вклучување на веб апликацијата.

2.4.3 Работа со податоци во сервлетски опсег (ServletConfig)

При креирање на инстанца од еден сервлет, веб контејнерот креира и уште еден дополнителен објект како инстанца од класата `ServletConfig` во кој сместува референци кон сите конфигурациски параметри за сервлетот поставени во конфигурациската датотека `web.config`. Овие параметри се чуваат во формат име-вредност и се однесуваат само на сервлетот во кој се дефинирани (додаток [Б.4](#)).

Објектот `ServletConfig` се пренесува како аргумент на `init()` методот при иницијализација на сервлетот од страна на контејнерот, па затоа преку него сервлетот може да пристапи до своите конфигурациски параметри. Овие параметри се читаат од конфигурациската датотека само еднаш при креирање на сервлетот и нивната вредност не може да се менува во кодот. Конфигурациските параметри се достапни само за сервлетот за кој се дефинирани и не можат да се споделуваат со останатите сервлети. Сепак, во пракса, параметрите на сервлетот ретко се користат, но објектот `ServletConfig` има референца кон апликацискиот контекст (`ServletContext`) кој е од големо значење.

Пристап до `ServletConfig`

За да се добие инстанца од `ServletConfig` потребно е се повика методот `getServletConfig()` со потпис: `public ServletConfig getServletConfig();`

Откако ќе се добие инстанцата, за пристап до конфигурациските податоци на сервлетот се повикува методот `getInitParameter` од интерфејсот `ServletConfig`:
`public String getInitParameter(String name)` Како аргумент на методот се пренесува името на параметарот сместено помеѓу ознаките `<param-name>`, а како одговор се добива неговата вредност поставена помеѓу ознаките `<param-value>` во `web.config`. Без разлика за каков тип на параметар станува збор, методот секогаш враќа текстуален тип, па затоа, ако се чита вредност која не е текст, потребно е таа да се конвертира (анг. cast) во соодветен тип.

`ServletConfig` го нуди методот `getInitParameterNames()` со чија помош се добиваат имињата на сите параметри до кои потоа може да се пристапи поединечно:
`public Enumeration getInitParameterNames()`

Во кодниот исечокот [2.14](#) е даден пример во кој при иницијализацијата на сервлетот `login` се чита и печати параметарот со име `login-attempts`, а потоа се читаат имињата на сите параметри на сервлетот и преку нив се пристапува до секој параметар поединечно, за на крајот да се испечати и името и вредноста на параметраот.

```

1  @WebServlet(name = "login", value = "/login")
2  public class LoginServlet extends HttpServlet {
3      public void init() {
4          ServletConfig config = getServletConfig();
5          String loginAttempts = config.getInitParameter("login-attempts");
6          System.out.println("The current value of login-attempts is " +
7              loginAttempts);
8          Enumeration<String> configParameters =
9              config.getInitParameterNames();
10         while (configParameters.hasMoreElements()){
11             String paramName = configParameters.nextElement();
12             System.out.println("param-name: " + paramName + " param-value: "
13                 + config.getInitParameter(paramName));
14         }
15     }
16 }

```

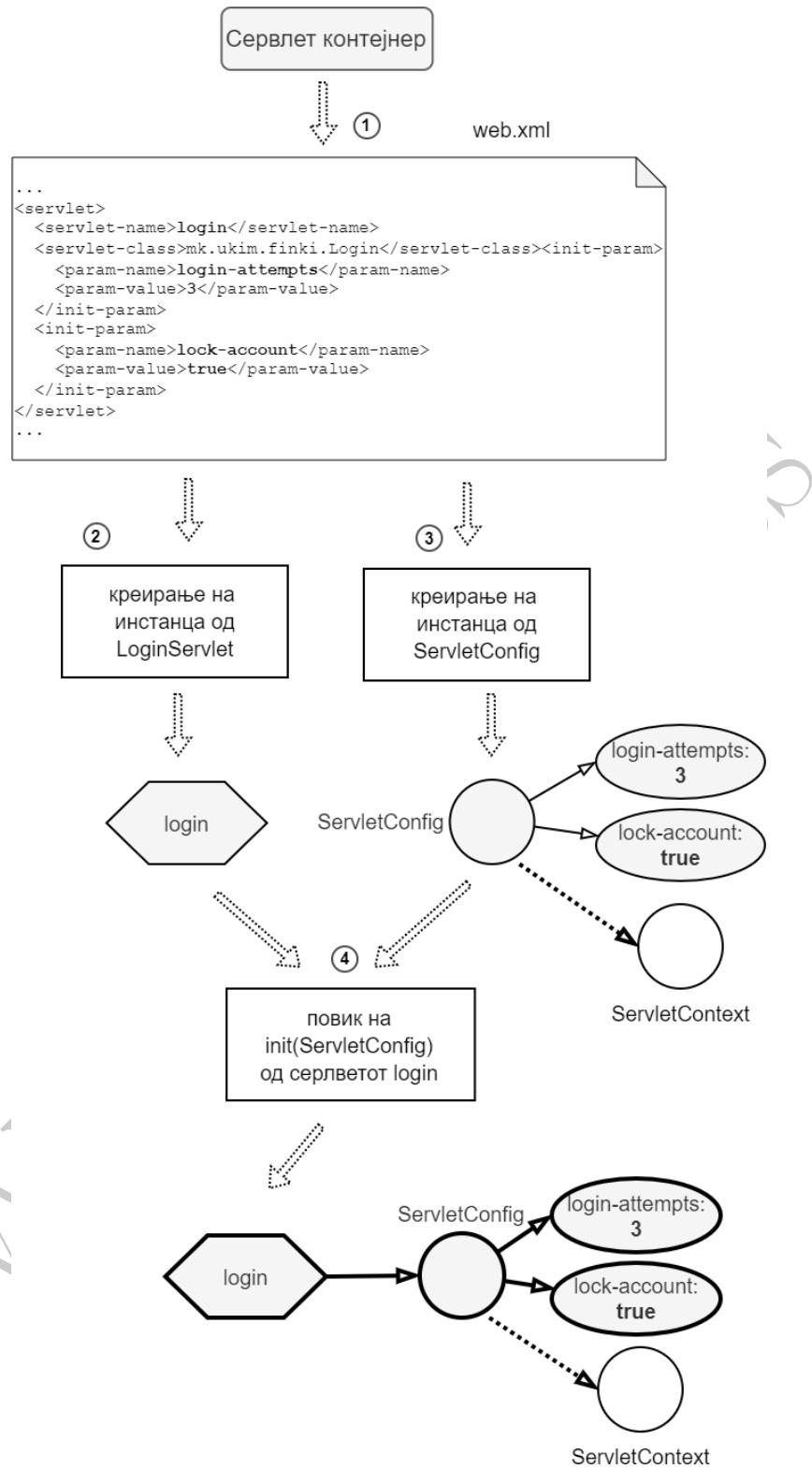
Изворен код 2.14: Пример за читање на параметри на сервлет

Преглед на параметри во серверски опсег

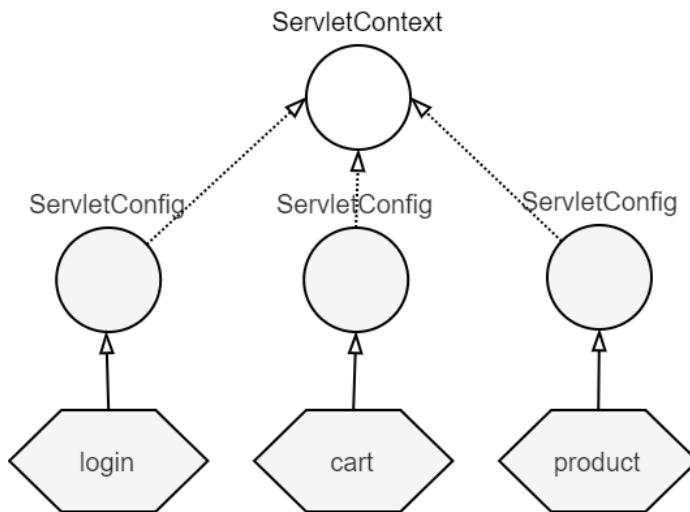
На слика 2-12 е дадена илустрација на дел од животниот циклус на `ServletConfig`, кој содржи референци кон параметрите на сервлетот кои ги поставува сервлет контејнерот од `web.xml` при иницијализација на сервлетот.

Објектот `ServletConfig` е дел од секој сервлет, па затоа неговото креирање е поврзано со креирањето на сервлетот на кој се однесува:

1. Сервлет контејнерот го чита `web.xml` и креира инстанца од сервлет
2. Сервлет контејнерот креира инстанца од `ServletConfig` и во неа поставува референци кон исчитаните параметри за сервлетот прочитани од `web.xml`. Сервлото контејнерот додава и референца кон `ServletContext` со цел сервлетот да може да пристапи до контекстот
3. Сервлет контејнерот го повикува мотедот `init(ServletConfig)` и воспоставува врска помеѓу сервлетот и `ServletConfig`. Овој метод е дел од животниот циклус на сервлетот, и ако се препокрие, во него програмерот веќе може да ги прочита сите параметри.



Слика 2-11: Преглед на животен век на **ServletConfig** и неговите параметри



Слика 2-12: Преглед на животен век на `ServletConfig` и неговите параметри

2.4.4 Работа со параметри и атрибути во опсег на барање

Пристап до параметри на барање

HTTP бараќата кои пристигнуваат од клиентите до контејнерот се конвертираат во објекти од типот `HttpServletRequest` кои потоа се предаваат на сервлетите. Овие објекти ги содржат сите елементи на барањето, меѓу кои и пренесените параметри до кои може да се пристапи преку методи од интерфејсот `HttpServletRequest`:

- `1 public String getParameter(String name)`
- `2 public Enumeration getParameterNames()`

Првиот метод го враќа параметарот со дадено име кој се пренесува во барањето како параметар во неговата URL адреса или во неговото телото како дел од некоја форма. Параметрите може само да се читаат и секогаш пристигнуваат од клиентите или од друг сервер во случај на препраќање.

На пример, ако URL адресата на барањето е `"/products/software?page=2"` и ако `request` е инстанца од `HttpServletRequest` која го содржи барањето, тогаш `request.getParameter("page")` ќе ја врати вредноста 2 како податок од типот `String`.

Досега, во неколку наврати видовме пристап до параметри од POST барање кои се пренесуваат во неговото тело. На пример, ако html страница содржи елемент за внесување податоци `<input type="text" name="user">` и ако тој елемент е дел од форма кој повикува сервлет, тогаш вредноста внесена од корисникот во тоа поле може да се прочита од барањето во сервлетот со `request.getParameter("user")`.

Управување со атрибути на барање

Објекти од типот `HttpServletRequest` може да се користат и за пренесување на кориснички дефинирани објекти во рамките на апликацијата. Слично како и кај контекстните атрибути, така и кај атрибутите на барањето се поставува име и вредност која може да биде било кој објект. Атрибутите, како објекти, се дел од барањето `HttpServletRequest` и опстануваат сè додека постои барањето. Тие постојат само во рамките на веб апликацијата на серверлот (во контејнерот) и не може да се пренасат назад до клиентите. Типично, атрибутите се поставуваат во барањата само во случаи кога тие се препраќаат на друг сервлет или ресурс (`jsp`) за понатамошна обработка со `RequestDispatcher`. Ако не се проследуваат до друг сервлет или ресурс, нивната примена нема смисла бидејќи по завршување на методот `service()`, контејнерот го брише барањето `HttpServletRequest`.

Слично како и кај контекстот `ServletContext`, методите за управување со атрибути на интерфејсот `HttpServletRequest` се:

```
1 public void setAttribute(String name, Object object)
2
3 public Object getAttribute(String name)
4
5 public void removeAttribute(String name);
```

Во кодниот исечок 2.15 е даден пример за поставување и читање на атрибути во барање. Во сервлетот `login` се чита само вредноста на лозинка, која заедно со корисничкото име ги испратил корисникот преку POST барање. Ако лозинката има вредност `spring`, се креира случајна вредност од типот `boolean` со име `luckyUser` која се доделува како атрибут на барањето. Потоа барањето се препраќа на сервлетот `home` кој го презема објектот на барањето заедно со атрибутот. Сервлетот го чита параметарот `username` испратен од корисникот и атрибутот `luckyUser` додаден во барањето од сервлетот. За да се добие вредноста на атрибутот, мора да се направи конверзија во типот `boolean` бидејќи атрибутот се внесува во барањето како објект од типот `Object`. Корисничкото име го испишува во одговорот, а во зависност од вредноста на атрибутот испишува и дополнителна порака. Атрибутот ќе биде валиден додека опстојува барањето, односно, неговата вредност ќе опстојува во сите следни сервлети и ресурси кон кои ќе се препрати истото барање. Ако корисникот генерира ново барање до сервлетот `login`, тогаш ќе се генерира нова вредност за атрибутот која на ниеден начин не е корелирана со претходната вредност.

```
1 @WebServlet("/login")
```

```

2 public class LoginServlet extends HttpServlet {
3     protected void doPost(HttpServletRequest request, HttpServletResponse
4             response) throws IOException, ServletException {
5         String password = request.getParameter("password");
6         if (password.equals("spring")){
7             Random random = new Random();
8             request.setAttribute("luckyUser", random.nextBoolean());
9             RequestDispatcher dispatcher =
10                 request.getRequestDispatcher("home");
11             dispatcher.forward(request,response);
12         }
13     }
14     @.WebServlet("/home")
15     public class HomeServlet extends HttpServlet {
16         protected void doGet(HttpServletRequest request, HttpServletResponse
17             response) throws IOException, ServletException {
18             response.setContentType("text/html");
19             PrintWriter out = response.getWriter();
20             out.println("Welcome " + request.getParameter("username"));
21             Boolean luckyUser = (Boolean) request.getAttribute("luckyUser")
22             if (luckyUser)
23                 out.println("You are winner of a free coupon! ");
}

```

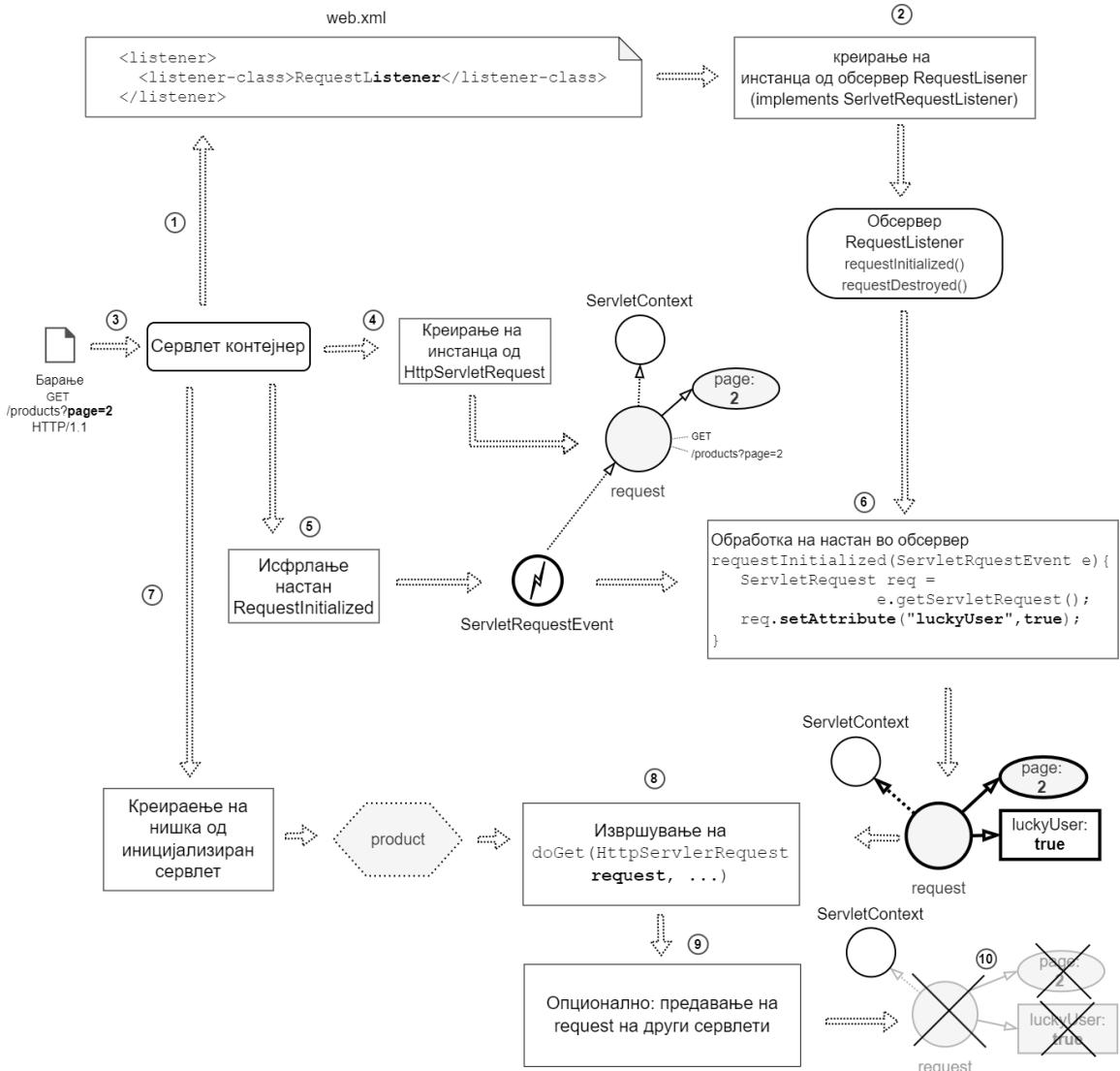
Изворен код 2.15: Пример за параметри и атрибути кај барање

Преглед на настани, параметри и атрибути кај опсег на барање

Како и кај контекстот, така и кај барањето се емитуваат настани при промена на неговата состојба. Настаните кои се важен дел од животниот циклус на едно барање се негова иницијализација од страна на контејнерот во моментот кога ќе пристигне и пред да биде доделено на сервлет за обработка, како и негово терминирање што се случува во моментот кога тоа ќе биде обработено, пред да се испрати одговорот на веб серверот. Во ваквите моменти контејнерот емитува настан кој е инстанца од типот `ServletRequestEvent`. Настанот нуди пристап и до барањето, а преку него и до контекстот. За обработка на настаните потребно е да се дефинира и регистрира обсервер. Оваа менацирана класа го имплементира интерфејсот `ServletRequestListener` кој содржи методи кои ќе се извршат при иницијализирање и терминирање на барање. Подетален опис за настаните

поврзани со барањето е даден во додаток [Б.7](#).

На слика [2-13](#) е дадена илустрација на животниот циклус на едно барање, неговата врска со параметрите и атрибутите и начинот на обработка на настаните поврзани со промена на неговата состојба.



Слика 2-13: Преглед на HttpServletRequest и неговите параметри и атрибути

1. Уште при вклучувањето на веб апликацијата, како и останатите менацирани компоненти, врз основа на `web.xml` серверт, контејнреот иницијализира инстанца од обсерверот за настани поврзани со барањата кој го имплементира интерфејсот `ServletRequestListener`. Ова се прави само на почетокот, во случај компонентата да е регистрирана, а потоа обсерверот ги обработува настаните во текот на целиот животен век на апликацијата

2. Клиентот испраќа HTTP барање кое веб серверот му го проследува на сервлет контејнерот. Барањето содржи параметар во неговата низа за пребарување
3. Сервлет контејнерот креира инстанца од `HttpServletRequest` врз основа на барањето (креира и инстанца од `HttpServletResponse` која намерно е изоставена на сликата). Барањето има референца кон параметрите од низата на пребарување, но исто така и кон сите податоци кои се пренесуваат во телото на барањето во случај на POST барање. Сервлет контејнерот поставува и референца кон сервлет контекстот
4. Бидејќи барањето е креирано, сервлет контејнерот исфрла настан во форма на инстанца од `ServletRequestEvent`. Оваа инстанца има референца кон барањето која може да се искористи во методот за обработка на настанот за да се пристапи до барањето
5. Обсерверот го извршува методот `requestInitialized(ServletRequestEvent e)` во кој се поставува атрибут на барањето. До барањето се пристапува преку постоечката референца во рамките на настанот.
6. Сервлет контејнерот креира нишка од сервлетот мапиран за URL адресата на барањето и го извршува соодветниот метод во зависност од типот на барањето. Во примерот, станува збор за методот `doGet()` на кој како аргумент се пренесуваат објектите за барањето и одговорот, па затоа може да се манипулира со атрибутите на барањето и да се читаат неговите параметри
7. Опционално, сервлетот може да го препрати барањето на друг сервлет за понатамошна обработка со сите негови параметри и атрибути.
8. Откако барањето ќе биде обработено од сите сервлети, сервлет контејнерот го брише заедно со неговите параметри и атрибути. Пред комплетно да ги ослободи сите ресурси, контејнрот повторно исфрала настан кој ќе биде обработен од обсерверот, но во него ќе биде повикан методот `requestDestroyed(ServletRequestEvent e)`. Со тоа завршува комплетниот животен циклус на барањето.

2.4.5 Работа со атрибути во сесиски опсег

При користење на една веб апликација, потребно е да се одржува постојана врска со клиентите, односно, мора да постои начин според кој апликацијата ќе го идентификува секој клиент при обработка на неговите барања за да генерира содржина поврзана со тој клиент. Апликациите мораат да одржуваат податоци за сесијата која ја иницираат преку првата интеракција со апликацијата.

Во едно типично сценарио, клиентот се најавува со своето корисничко име и лозинка, а потоа, при секое следно барање од корисникот и навигација низ

страниците, веб апликацијата има информации за него бидејќи одржува податоци за неговата започната сесија откако се најавил. Но, како е возможно веб апликацијата да чува информации за корисниците кога целта комуникацијата помеѓу нив се одвива со помош на протоколот HTTP, кој по природа е безсостојбен протокол? Со ваквата комуникација, не постои разлика дали две барања потекнуваат од еден клиент или од два различни клиенти. Според тоа, како веб апликацијата ги разликува различните клиенти? Одговорот на ова прашање веќе го дадовме во секција 1.8 каде ја описуваме употребата на колачини. И покрај тоа што HTTP е безсостојбен протокол, колачината овозможуваат веб серверот и прелистувачот да си разменуваат информации преку кои ќе се идентификуват различните корисници. Така, при првата посета на веб страница, веб серверот може да испрати колаче со уникатен број за корисникот кое ќе биде зачувано во меморијата на прелистувачот и ќе биде испраќано назад до серверот при секое следно барање упатено кон него.

Но, што ако станува збор за веб апликација, а не веб страница каде со колачината управува веб серверот? Во тој случај, самата апликација треба да ги направи следните операции

- да креира посебен простор во меморијата т.н. сесиски објект во кој ќе чува кориснички дефинирани податоци
- да креира колаче со уникатен број за идентификација на тој објект (мора да води сметка да нема повторување на две различни колачини)
- да постави животен век на колачето кое ќе го определи траењето на самата сесија,
- да го вметне колаченто во одговорот,
- за секое пристигнато барање да проверува дали постои колаче и да го искористи идентификацискиот број од колачето за да пристапи до сесискиот објектот од кој ќе ги прочита податоците за клиентот.

Сето ова укажува на фактот дека управувањето со колачини и сесиски објекти од страна на самите програмери значително би го отежнало креирањето на веб апликации. Затоа, кај веб апликациите од Јава серверското издание, оваа обврската комплетно ја презема сервлет контејнерот кој креира и одржува сесиски објекти од типот HttpSession, но и овозможува едноставен начин за управуваме со нив.

Пристап до сесиски објект

За да се креира сесиски објект HttpSession за даден корисник потребно е да се повика еден од методи на интерфејсот HttpServletRequest:

```
1 public HttpSession getSession();  
2 public HttpSession getSession(boolean create);
```

Пријот метод проверува дали постои сесиски објект во контејнерот. Тоа го постигнува преку проверка дали барањето содржи колаче со име JSESSIONID. Ако не постои такво колаче, одното, ако не постои сесија за корисникот, креира сесиски објект со уникатен сесиски идентификатор (12 цифrena хексадецимална вредност), го назначува објектот на барањето, креира колаче со име JSESSIONID чија вредност е сесискиот идентификатор и го вметнува колачето во одговорот. На крајот, методот враќа референца кон новиот сесискиот објект. Ако во барањето постои колаче, а со тоа и валиден сесиски објект за прочитаниот сесиски идентификатор, тогаш методот само ја враќа референцата кон постоечкиот сесискиот објект. Според тоа, до сесискиот објект може да се пристапи од било кој сервер на апликацијата сè додека барањето содржи колаче со валиден сесиски идентификатор.

Методот може да се повика и со аргумент од типот `boolean` и во тој случај нова сесија се креира во зависност од неговата вредност. Ако се повика `getSession(true)`, се добива идентично однесување како и `getSession()`. Ако се повика `getSession(false)` и ако не постои сесија поврзана за барањето, тогаш нема да се креира нова сесија и методот ќе врати `null`. Ако постои сесија, тогаш методот ја враќа нејзината референца. Овој случај се користи за да се пристапи само до веќе постоечки сесиски објект, без да се креира нов ако не постои.

Ако клиентот не поддржува колачиња, на пример, постави во неговиот прелистувач да не ги прифаќа, тогаш при секој повик на `getSession()` ќе се креира нова сесија. Ако пак клиентот користи анонимен пристап (анг. incognito access), тогаш колачето ќе се чува и испраќа во барањата сè додека е активна анониманта сесија, а со тоа ќе функционира и работата со сесии. Откако корисникот ќе излезе од ваквиот начин на работа, колачето ќе биде избришано од меморијата на прелистувачот и при следниот обид сесијата за тој клиент нема да биде валидна.

Управување со сесиски атрибути

Сесискиот објект овозможува споделување на атрибути за корисниците во рамките на целата апликација. За управување со атрибутите на сесија се користат истоимени методи како и за управување со контекстните атрибути, но кај сесиите, тие се дел од интерфејсот `HttpSession`:

```
1 public void setAttribute(String name, Object object)  
2  
3 public Object getAttribute(String name)
```

```

4
5 public void removeAttribute(String name)

```

За разлика од `ServletContext` и `ServletConfig`, сесиските објекти не содржат параметри кои се поставуваат во конфигурациска датотека.

Во кодниот исечок 2.16 е даден пример за поставување и читање на атрибути во сесиски објект. Во сервлетот `login` се читаат вредностите на корисничкото име и лозинка кои ги испратил корисникот преку POST барање и доколку лозинката има вредност `spring`, се креира нов сесиски објект во кој се сместува корисничкото име. Потоа корисникот се пренасочува кон сервлетот `home`. Иако станува збор за поедноставне пример, описанот сервлет има слична улога како и сервлетите за најава кај продукцијски апликации каде по успешната најава на корисникот се креира сесија за него во која се сместуваат податоци кои го идентификуваат корисникот во текот на животниот век на сесијата при навигацијата низ целата апликација. Во вториот сервлет `home`, најпрво се проверува дали сесисикот објект е нов, односно дали повикот на `getSession()` креирал нов објект или објектот веќе постоел. Ако станува збор за нова сесија, тоа значи дека корисникот не поминал низ процесот на најава, па затоа повторно се враќа на страницата `login.html`, чија содржина не е дадена во кодот, но содржи полиња за корисничко име и лозинка чии вредности се испраќаат до сервлетот `login`. Ако сесискиот објект веќе постоел, тогаш од него се чита корисничкото име кое било претходно зачувано од сервлетот `login` и се запишува во одговорот.

```

1 @WebServlet("/login")
2 public class LoginServlet extends HttpServlet {
3     protected void doPost(HttpServletRequest request, HttpServletResponse
4             response) throws IOException, ServletException {
5         String username = request.getParameter("username");
6         String password = request.getParameter("password");
7         if (password.equals("spring")){
8             HttpSession session = request.getSession()
9                 session.setAttribute("user", username)
10                response.sendRedirect("home");
11        }
12    }
13 @WebServlet("/home")
14 public class HomeServlet extends HttpServlet {
15     protected void doGet(HttpServletRequest request, HttpServletResponse
16             response) throws IOException, ServletException {

```

```

16     HttpSession session = request.getSession()
17     if (session.isNew())
18         response.sendRedirect("/login.html");
19     }
20     else {
21         response.setContentType("text/html");
22         PrintWriter out = response.getWriter();
23         HttpSession session = request.getSession();
24         String username = (String) session.getAttribute("user");
25         out.println("Welcome " + username)
26     }
27 }
28 }
```

Изворен код 2.16: Пример за HttpSession

Ако ја повикаме URL адресата `"/login.html"` и ако во полињата за најава внесеме Admin и spring, тогаш ќе бидеме пренасочени на URL адресата `"/home"` и во прелистувачот ќе се појави страница со порака "Welcome Admin". Ако ја затвориме страницата или прелистувајќи го и ако повторно се обидеме да пристапиме директно до URL адресата `"/home"`, ќе се испечати истата порака. Како всушност серверот нè идентификува во вториот обид кога веќе еднаш сме ја затвориле страницата? При првата најава контејнерот креира колаче со сесиски идентификатор кое го вметнува во одговорот (роверете ја неговата содржина преку детеален преглед на HTTP одговорот во прелистувачот). Кога прелистувајќи го доби одговорот со пораката "Welcome Admin" го зачува добиеното колаче на диск, па истото колаче го испраќа во секое следно барање, вклучувајќи го и вториот обид за директен пристап до URL адресата `"/home"`. Сервлетот мапиран на оваа адреса при повик на `getSession()` го пребарува објектот со сесиски идентификатор добиен од колачето, и бидејќи таков објект постои, методот `isNew()` врќа вредност `false`, па затоа вредноста за корисничкиот име ќе биде успешно прочитана од сесикиот објект и вратена како одговор на барањето.

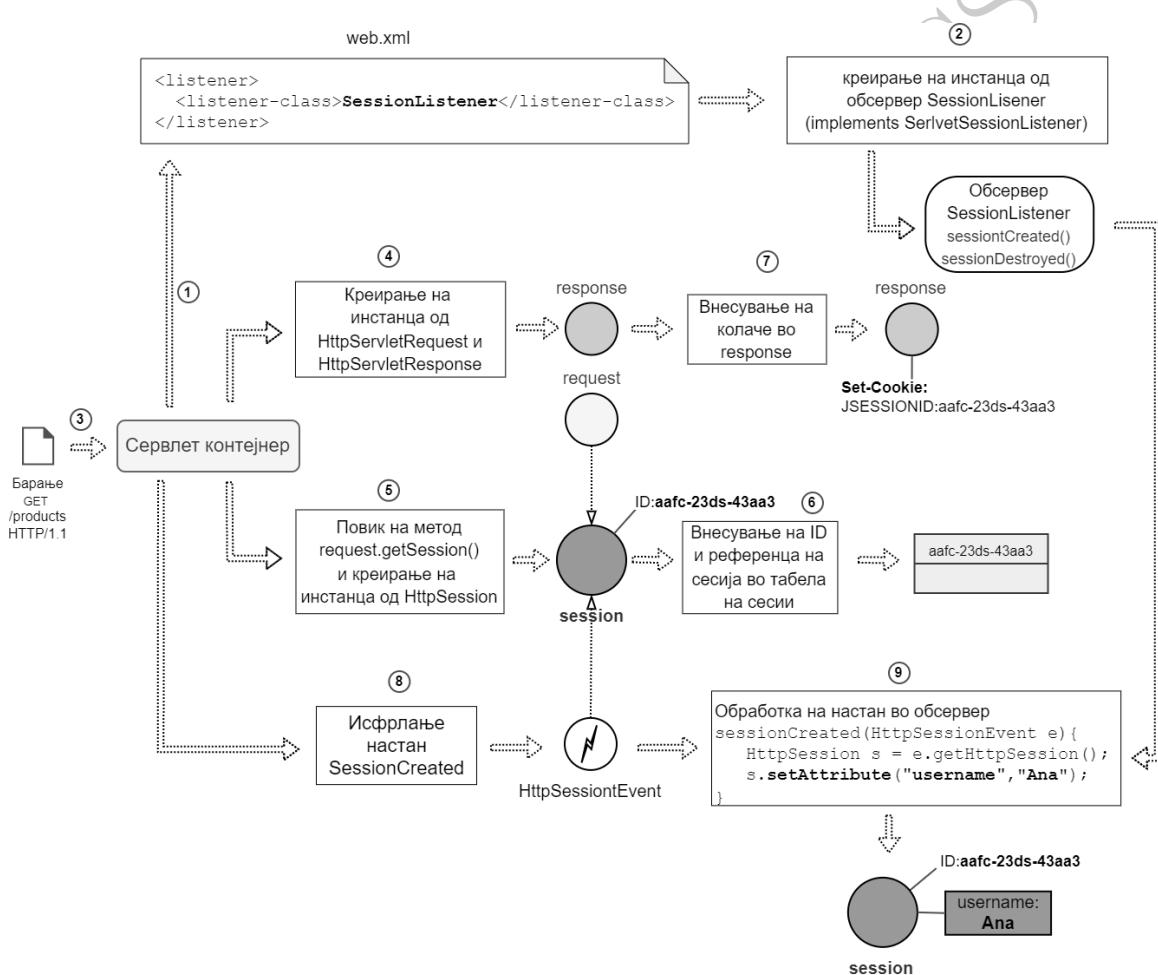
Ако ја повториме истата операција после еден час откако сесијата ќе истече или ако упатиме барање преку друг прелистувач, тогаш повикот на методот `getSession()` ќе врати нов сесиски објект, `isNew()` ќе врати вредност `true` и затоа корисникот ќе биде вратен назад на страницата за најава `login.html`.

Преглед на настани и атрибути во сесиски опсег

Аналогно на досегашните настани и обсервери кај контекстот и барањето, постојат настани кои се емитуваат при промена на состојбата на сесијата или при

промена на сесиските атрибути. Кога станува збор за промена на состојбата на сесијата, се еmitува настан кога сесијата се криера и кога сесијата се затвора. Настанот е инстанца од типот `HttpSessionEvent` и нуди пристап до сесискиот објект. За обработка на настаните потребно е да се дефинира и регистрира обсервер кој го имплементира интерфејсот `HttpSessionListener` кој содржи методи за иницијализација и терминирање на сесијата. Подетален опис за настаните поврзани со промената на состојбата на сесијата и состојбата на атрибутите е даден во додаток [Б.10](#).

На слика [2-14](#) е дадена илустрација на животниот циклус на една сесија, управувањето со атрибути и дел од настаните поврзани со неа. На сликата е прикажано сценариото во случај кога корисникот за прв пат пристапува до веб апликацијата.



Слика 2-14: Преглед на `HttpSession` и неговите атрибути

- Уште на самиот почеток на животниот циклус на веб апликацијата, кога се вчитуваат менацираните компоненти, доколу е дефиниран се вчитува обсервер кој ги обработува настаните поврзани со промената на состојбата на

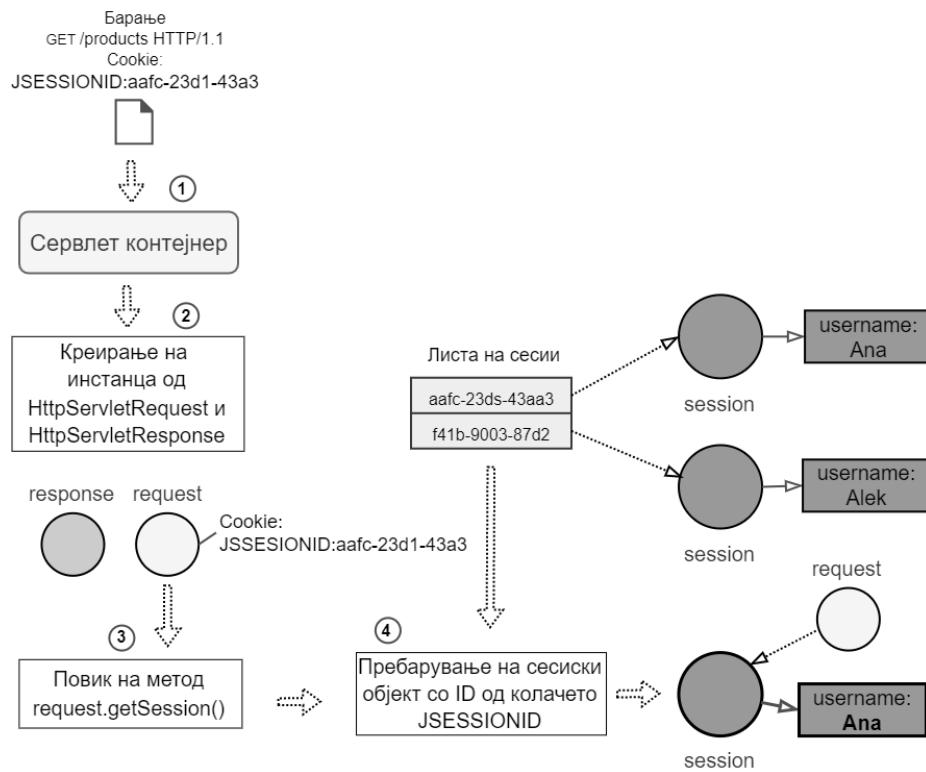
сесијата. Обсерверот го имплементира интерфејсот `ServletSessionListener` кој содржи методи кои се повикуваат при креирање или терминирање на сесија.

2. Откако ќе дојде барање од клиентот, сервлет контејнерот креира инстанца од `HttpServletRequest` и `HttpServletResponse`.
3. При обработка на барањето во одреден сервлет, се повикува методот `request.getSession()`. Бидејќи станува збор за **прво барање** на корисникот, сервлет контејнерот креира инстанца од `HttpSession` со уникатен идентификатор во рамките на целата апликација. Сервлетот поставува и референца кон оваа инстанца во самото барање за да може да се пристапи до сесијата при обработка на барањето во сервлетите (барањето се пренесува како аргумент на методите за негова обработка).
4. сервлет контејнерот го внесува идентификаторот на сесијата во посебна структура во меморијата и во истата внесува референца кон креираната инстанца од сесискиот објект. Оваа структура ја користи за во иднина да го лоцира сесискиот објект врз основа на неговиот идентификатор.
5. Серверот поставува колаче во одговорот со име `JSESSIONID` и вредност која одговара на идентификаторот на сесискиот објект. Откако одговорот ќе биде претворен во HTTP одговор и испратен до корисникот, неговиот прелистувач ќе го зачува поставеното колаче и истото ќе го испраќа до серверот во секое наредно барање.
6. Бидејќи сесијата е креирана, сервлет контејнерот исфрла настан преку инстанца од `HttpSessionEvent`. Овој објект има референца кон сесијата со цел да може да се пристапи до неа при обработка на настанот.
7. Обсерверот го извршува методот ⁸`sessionCreated(HttpSessionEvent e)` во кој се поставува атрибут на сесијата. До сесијата се пристапува преку постоечката референца во рамките на настанот.
8. Откако барањето ќе биде обработено и терминирано, сесискиот објект, заедно со сите негови поставени атрибути продолжува да егзистира во рамките на сервлет контејнерот.

На слика 2-15 е дадена илустрација за сценариото во случај кога корисникот веќе пристапил до ресурс на веб апликацијата кој креирал сесиски објект и како резултат на тоа испратил колаче до прелистувачот. Според сликата, до апликацијата пристапил и друг корисник за кој е веќе креирана нова инстанца од сесискиот објект со различен идентификатор.

1. Корисникот испраќа ново барање до веб апликацијата, но овој пат неговиот

⁸Не е задолжително да се регистрира обсервер за сесија, но во примерот служи за иницијализација на атрибут пред било кое барање да пристапи до сесијата



Слика 2-15: Преглед на HttpSession и неговите атрибути

предизвикава испраќа колачето со име JSESSIONID

2. Сервлет контејнерот қреира инстанца од HttpServletRequest во која, како и сите останати заглавја на барањето, се поставува и пренесеното колаче.
3. При обработка на барањето од страна на сервлет, се повикува методот `request.getSession()`. Сервлет контејнерот ја користи вредноста на колачето JSESSIONID за да ја пребара сесијата идентификувана со таа вредност во својата листа на сесии. Откако ќе ја лоцира, ја враќа нејзината референца и со тоа сервлетот кој го обработува барањето добива пристап до сесискиот објект за дадениот корисник заедно со неговите претходно поставени атрибути.

2.4.6 Генерален преглед на параметри, атрибути и опсези на важење

По целокупното детално поединечно представување на опсезите на важење на параметрите и атрибутите, во продолжение ќе биде даден целосен преглед на сервлет контејнер со сите опсези на важење и објектите со кој тој управува.

На слика 2-16 се прикажани компонентите за секој од опсезите на важање.

Централната улога во сервлет контејнерот ја има `ServletContext` бидејќи не-

говиот опсег на важење е глобален, односно кај податоците сместени во контекстот, опсегот на важење е целата апликација. До него може да се пристапи во од било кој сервлет преку `Servletconfig` или пак од било која компонента на која ѝ се предава барање. Контекстните податоците се или параметри поставени во конфигурациската датотека `web.xml` или атрибути кои се поставуваат во самиот код. Барањата од различни клиенти и барањата за различни ресурси ги добиваат истите вредности за параметрите и атрибутите зачувани во контекстот.

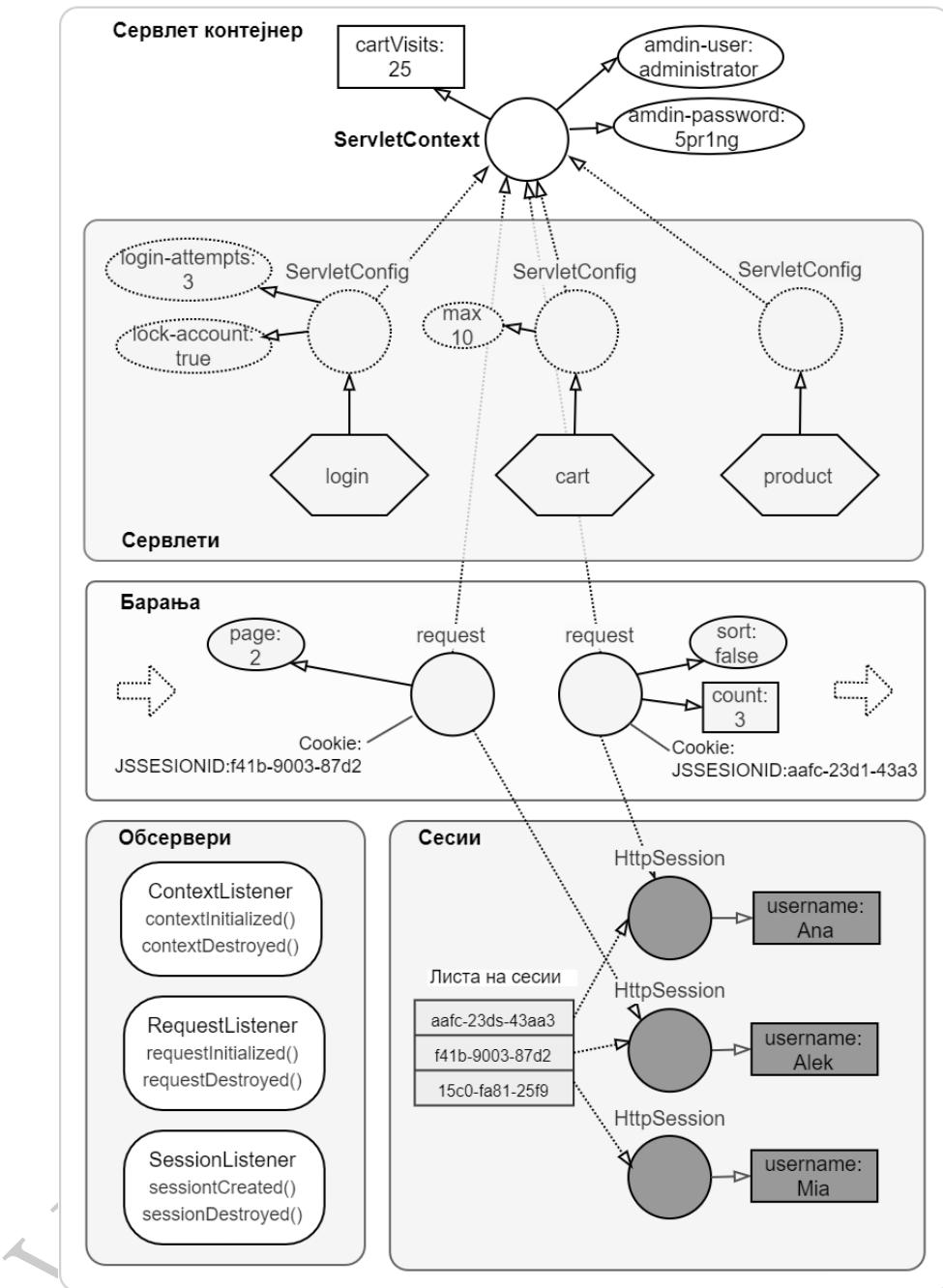
Податоците зачувани во `ServletConfig` важат за сите барања кои ги обработува сервлетот на кој се однесува. Податоците зачувани во овој објект можат да бидат само од типот параметри, па затоа не можат да се менуваат во кодот. Нивната вредност се поставува или во конфигурациската датотека `web.xml` или во анотацијата `@WebServlet` при дефиниција на класата за сервлетот. Секое различно барање до еден сервлет ги добива истите вредности на параметри зачувани за сервлетот.

Каде податоците сместени во барањето, опсегот на важење е самото барање. Податоците се пренесуваат низ оние сервлети од апликацијата на кои се предава барањето. Податоците во барањето се или параметри поставени од клиентите или атрибути кои се поставуваат во самиот код. Секое различно барање може да има различни параметри во зависност од ресурсот што го побарува клиентот и да добие различни објекти за атрибутите во зависност од тоа која компонента го обработува. Барањата имаат најкраток животен век и привремен карактер бидејќи постојат само додека апликацијата ги обработува, по што се бришат.

Податоците зачувани во сесија важат само за еден клиент (од еден прелистувач) на ниво на цела апликација, без разлика на ресурсот кој се побарува преку барањето или сервлетот кој го опслужува. Тие можат да бидат само од типот атрибути и се поставуваат во кодот на апликацијата. Секое различно барање до било кој сервлет кое пристигнува од истиот клиент ги добива истите вредности зачувани за неговата сесија. Сесиите за различни корисници остануваат во сервлет контекстот додека не се избираштат експлицитно во кодот или додека не истече нивниот поставен животен век.

2.4.7 Вежба: Работа со параметри и атрибути во опсези на апликација

За да направиме преглед на работата со параметри и атрибути од сите опсези на важење, ќе изработиме едноставна апликација која ќе ни помогне да ги согладаме нивните вредности во различни кориснички сценарија. Креирајте празен веб проект и во именикот `src/main/webapp` креирајте датотека со име `index.html` и содржина како во кодниот исечок 2.17.



Слика 2-16: Преглед на елементи на сервлет контејнер со параметри и атрибути на различни опсези

```

1 <html>
2   <body>
3     <a href="/s1?page=1">Servlet 1</a><br>
4     <a href="/s2?page=2">Servlet 2</a><br>

```

```

5      <a href="/s3">Servlet 3</a><br>
6    </body>
7 </html>
```

Изворен код 2.17: Содржина на index.html датотека за повикување сервлети

Преку html датотеката се генерираат барања до три различни URL адреси кои ќе повикуваат три различни сервлети. Првите два линка во барањето пренесуваат и параметар во URL адресата со име `page` и вредност 1, односно 2.

Во именикот за складирање на класи, креирајте ги класите дадени во кодниот исечок 2.18.

Класата `MyContextListener` го имплементира интерфејсот за обработка на настани поврзани со промена на состојбата на контекст. Во методот `contextInitialized()` кој се повикува еднаш при инцијализација на контекстот, се зема референца до контекстот и во него се поставува атрибут со име `app-counter` и почетна вредност 0.

Класата `MySessionListener` го имплементира интерфејсот за обработка на настани поврзани со промена на состојбата на корисничка сесија. Во методот `sessionCreated()` кој се повикува при креирање на нова сесија за корисник (кога се повикува методот `getSession()` и не постои сесија за тој корисник), се зема референца до сесискиот објект и во него се поставува атрибут со име `user-counter` и почетна вредност 0.

Класата `MyRequestListener` го имплементира интерфејсот за обработка на настани поврзани со промена на состојбата на барање. Во методот `requestInitialized()` кој се повикува кога ќе се креираа објектот за барање врз основа на пристигнатото HTTP барање, се зема референца до тој објект и во него се поставува атрибут со име `req-counter` и почетна вредност 0.

```

1 @WebListener
2 public class MyContextListener implements ServletContextListener {
3   public void contextInitialized(ServletContextEvent event) {
4     ServletContext context = event.getServletContext();
5     context.setAttribute("app-counter", 0);
6   }
7   public void contextDestroyed(ServletContextEvent event) {
8   }
9 }
10 @WebListener
11 public class MySessionListener implements HttpSessionListener {
12   public class MySessionListener implements HttpSessionListener {
13     public void sessionCreated(HttpSessionEvent event) {
```

```

14     HttpSession session = event.getSession();
15     session.setAttribute("user-counter",0);
16 }
17 public void sessionDestroyed(HttpSessionEvent event) {
18 }
19 }
20 @WebListener
21 public class MyRequestListener implements ServletRequestListener {
22     public void requestInitialized(ServletRequestEvent event) {
23
24         ServletRequest request = event.getServletRequest();
25         request.setAttribute("req-counter",0);
26     }
27     public void requestDestroyed(ServletRequestEvent event) {
28     }
29 }
```

Изворен код 2.18: Имплементација на слушачи на настани за контекст сесија и барање

Во конфигурациската датотека `web.xml`, сместена во именикот `src/main/webapp/WEB-INF`, внесете го следниот код за дефиниција на параметар со име `size` и вредност 100 како дете на јазелот `<web-app>`:

```

1 <context-param>
2   <param-name>size</param-name>
3   <param-value>100</param-value>
4 </context-param>
```

Во пакетот за класи, креирајте класа со име `ScopeServlet1` и содржина како во кодниот исечок 2.19. Со помош на анотацијата `@WebServlet`, на сервлетот му се доделува името за внатрешна употреба `Serlvet1`, се пресликува во URL адресата `"/s1"` и му се доделува параметар кој ќе важи во рамките на сервлетот со име `pages` и вредност 10. Потоа, во имплементацијата на методот `doGet()` најпрво се земаат референци до конфигурацискиот објект за серфлетот `ServletConfig`, контекстот `ServletContext` и корисничката сесија `HttpSession`. Референцата за објектот кој го претставува барањето се пренесува како аргумент на самиот метод.

Во следниот дел, се читаат сите параметри за објектите кои ги поддржуваат. Контекстниот параметар поставен во `web.config` се чита од контекстот и се сместува во променливата `appSize`, параметарот на сервлетот со име `pages` поставен во анотацијата се сместува во променливата `configPages`, а параметарот кој се

чита од барањето (ако не постои, ќе се изврли грешка) и кој клиентот треба да го обезбеди или во низата за пребарување или телото на барањето се сместува во променливата `requestPage`.

Потоа, следни читање и уредување на содржината на атрибутите. Најпрво се чита контекстниот атрибут `app-counter`, се сместува во променливата `appCounter` чија вредност се инкрементира и повторно се сместува како нова вредност на контекстниот атрибут. Истата операција се повторува и за сесискиот атрибут `user-counter` и атрибутот на барањето `req-counter`. Нивните вредности пред да се инкрементираат се сместуваат во променливите `userCounter` и `requestCounter`. За потсетување, почетните вредности на сите три атрибути беа иницијализирани во слушачите на настани во кодниот исечок 2.18.

На крајот, името на сервлетот и паровите име-вредност на сите параметри и атрибути се печатат во одговорот кој се испраќа назад до клиентот.

```

1  @WebServlet(name = "Servlet1", value = "/s1",
2      initParams = {@WebInitParam(name = "pages", value = "10")})
3  public class ScopeServlet1 extends HttpServlet {
4      public void doGet(HttpServletRequest request, HttpServletResponse
5          response) throws IOException {
6          ServletConfig config = getServletConfig();
7          ServletContext context = getServletContext();
8          HttpSession session = request.getSession();
9
10         int appSize = Integer.parseInt(context.getInitParameter("size"));
11         int configPages = Integer.parseInt(config.getInitParameter("pages"));
12         int requestPage = Integer.parseInt(request.getParameter("page"));
13
14         int appCounter = (int)context.getAttribute("app-counter");
15         context.setAttribute("app-counter", ++appCounter);
16         int userCounter = (int)session.getAttribute("user-counter");
17         session.setAttribute("user-counter", ++userCounter);
18         int requestCounter = (int)request.getAttribute("req-counter");
19         request.setAttribute("req-counter", ++requestCounter);
20
21         response.setContentType("text/html");
22         PrintWriter out = response.getWriter();
23         out.println("<html><body><h4>" + config.getServletName() + "</h4>");
24         out.println("Parameters: appSize = " + appSize + " configPages =
25             " + configPages + " requestPage = " + requestPage + "<br>");
26         out.println("Attributes: appCounter = " + appCounter + " userCounter =
27             " + userCounter);
28     }
29 }
```

```

25      = "+userCounter+ " requestCounter = "+requestCounter);
26     out.println("</body></html>");
27 }

```

Изворен код 2.19: Имплементација на сервлет Servlet1

Во пакетот за класи, креирајте уште една класа со име `ScopeServlet2` со идентична содржина на методот `doGet()`. Единствено, овој сервлет треба да се разликува во вредностите во анотацијата `@WebServlet` и да има име за внатрешна употреба `Serlvet2`, URL адресата `"/s2"` и параметар со име `pages` и вредност 20.

Креирајте трета класа `ScopeServlet3` со име за внатрешна употреба `Serlvet3` и URL адресата `"/s3"` (без параметар). Имплементацијата на класата е прикажана во кодниот исечот 2.20. Сервлетот работи само со атрибутот на барањето `req-counter` така што ја чита неговата вредност, ја инкрементира, ја доделува новата вредност на атрибутот на барањето и го препраќа истото барање до сервлетот `Servlet1`, но при тоа го додава во низата за пребарување на барањето параметарот `page` со вредност 3.

```

1 @WebServlet(name = "Servlet2", value = "/s2",
2             initParams = {@WebInitParam(name = "pages", value = "20")})
3 public class ScopeServlet1 extends HttpServlet {
4     public void doGet(HttpServletRequest request, HttpServletResponse
5                        response) throws IOException {
6         //same code as doGet in Servlet1
7     }
8     @WebServlet(name = "Servlet3", value = "/s3")
9     public class ScopeServlet3 extends HttpServlet {
10        public void doGet(HttpServletRequest request, HttpServletResponse
11                           response) throws IOException, ServletException {
12            Integer requestCounter =
13                (Integer)request.getAttribute("req-counter");
14            request.setAttribute("req-counter", ++requestCounter);
15            RequestDispatcher rd = request.getRequestDispatcher("/s1?page=3");
16            rd.forward(request, response);
17        }
18    }

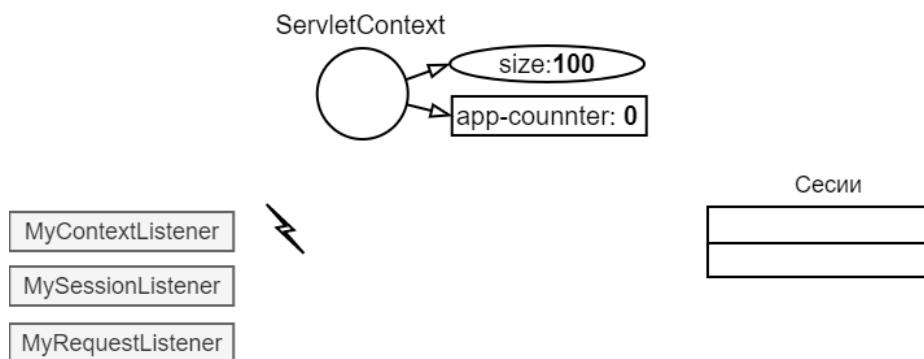
```

Изворен код 2.20: Имплементација на сервлети Servlet2 и Servlet3

Пред да ја извршите апликацијата, во конфигурацијата на интегрираниот веб сервер и контејнер (Tomcat) во IntelliJ поставете го апликацискиот кон-

текст на поставената апликација (Tomcat -> Edit Configurations... -> Deployment: Application context) да биде "/".

Извршете ја апликацијата и ќе ја добиете страницата со три линка. На сликата 2-17 е прикажана почетната состојба на апликацијата. Во овој момент се иницијализирани слушачите и е креиран контекстот, па затоа веќе се извршил методот `contextInxitialized()` од `MyContextListener` кој ја поставил вредноста на контекстниот атрибут `app-counter` на 0. Сервлетите се уште не се прочитани бидејќи се уште не е генерирано барање за ниту еден од нив.



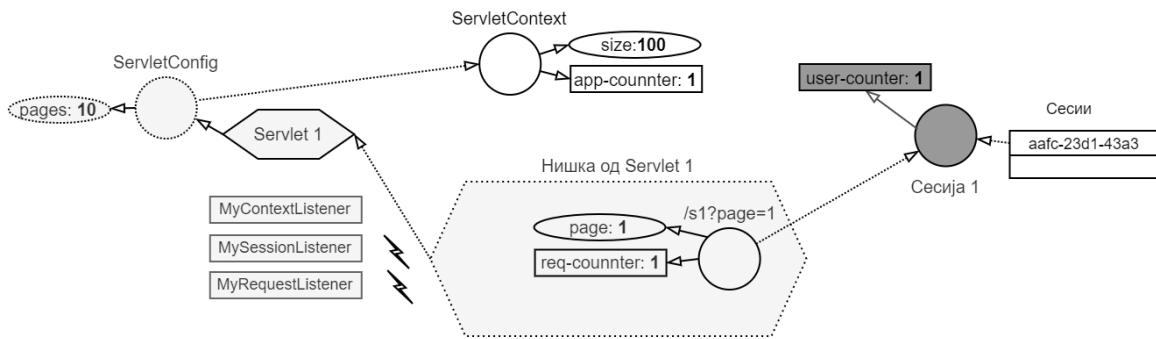
Слика 2-17: Почекна состојба на апликацијата

Кликнете на првиот линк кој ќе ве пренасочи на адресата `/s1?page=1`. Како одговор на барањето се добива порака со следната содржина:

- 1 Servlet1
- 2 Parameters: appSize = 100 configPages = 10 requestPage = 1
- 3 Attributes: appCounter = 1 userCounter = 1 requestCounter = 1

На сликата 2-18 е прикажана состојбата на сите компоненти, параметри и атрибути на апликацијата. Бидејќи побараната адреса е пресликана во сервлетот `Servlet1`, тој се иницијализира и се креира нишка за обработка на барањето. Дополнително, бидејќи е пристигнато ново барање се испалува настан за ново барање кој се обработува со методот `requestInitialized()` од слушачот `MyRequestListener` и со кој се поставува атрибутот `req-counter` на вредност 0. Овој настан ќе се испалува со секое следно барање. Во имплементацијата на методот `doGet()` во сервлетот се побарува референца до сесијата за барањето, но бидејќи иницијално барањето не содржи колаче, односно не постои сесија, се испалува настан откако таа ќе се креира и затоа се извршува методот `sessionCreated()` од слушачот `MySessionListener`. Со креирањето на сесијата се додава колаче во одговорот, па во секое следно барање од овој клиент, ќе се пренесува и колачето.

Comment: да се гргне cookie од барањето бидејќи е прво, па не постои

Слика 2-18: Состојба на апликацијата откако ќе се повика `"/s1?page=1"`

Од пораката се гледа дека е генерирана од сервлетот `Servlet1`. Вредноста на `appSize` е 100 бидејќи таа вредност беше поставена во `web.xml`. Истата вредност ќе се прикажува без разлика за кој сервлет станува збор, па затоа нема да ја коментираме во следните примери. Вредноста на `configPages` е 10 бидејќи станува збор за параметар на првиот сервлет поставен во неговата конфигурација. Вредноста на `requestPage` е 1 бидејќи URL адресата на самото барање е `"/s1?page=1"`. Сите атрибути имаат вредност 1 бидејќи станува збор за прво барање.

Кликнете на копчето за враќање наназад (анг. back) на прелистувачот и овој пат, одбоерете го вториот линк. Како одговор се добива порака од сервлетот `Servlet2`.

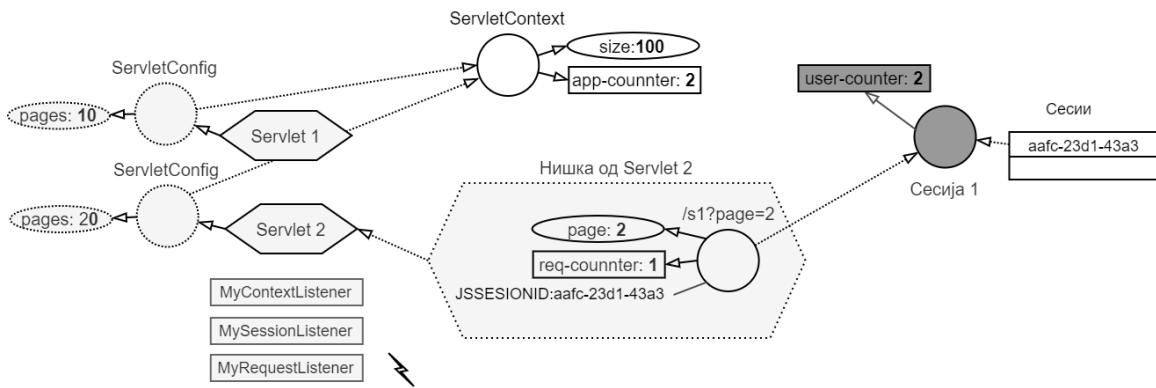
```

1 Servlet2
2 Parameters: appSize = 100 configPages = 20 requestPage = 2
3 Attributes: appCounter = 2 userCounter = 2 requestCounter = 1

```

На сликата 2-19 е прикажана состојбата на апликацијата откако ќе се опслужи барањето за адресата `"/s2?page=2"`. Ова барање го опслужува сервлетот `Servlet2` кој се иницијализира и се креира нишка за барањето. Овојпат, во имплементацијата на методот `doGet()` не се креира сесија бидејќи истата постои и се добива референца до неа за пристап до сесиските атрибути.

За разлика од претходниот пример, `configPages` е 20 бидејќи станува збор за параметар на сервлет со исто име како и претходно, но овој пат параметарот е поставен во конфигурацијата на `Servlet2`. Вредноста на `requestPage` е 2 бидејќи URL адресата на самото барање е `"/s2?page=2"`. Во однос на атрибутите, променливата `appCounter` со вредност на контекстен атрибут има вредност за еден поголема од претходната, односно 2. Контекстниот атрибут ќе се инкрементира при секој повик на `doGet()` методот, без разлика за кој корисник или сервлет (освен за `Servlet3` станува збор. Вредноста на `userCounter` која се однесува на сесискиот атрибут е 2, бидејќи станува збор за иста сесија, па затоа е за 1 пого-



Слика 2-19: Состојба на апликацијата откако ќе се повика `"/s2?page=2"`

лема од вредноста во претходното барање. Вредноста на `requestCounter` останува 1 бидејќи кај вториот повик станува збор за сосема ново барање за кое вредноста на атрибутот се поставува на 0 пред да се предаде на сервлетот за обработка во методот `requestInitialized()` од слушачот `MyRequestListener`. Секогаш кога барањето ќе биде обработено само од еден сервлет, вредноста на оваа променлива ќе биде 1, без разлика дали станува збор за ист корисник или сесија.

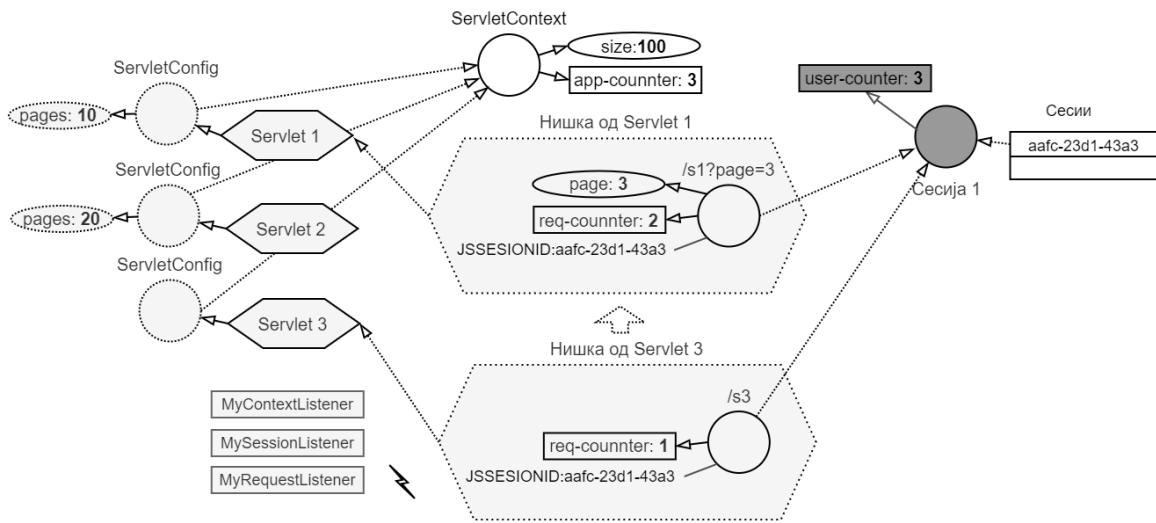
Кликнете на копчето за враќање наназад (анг. back) на прелистувачот и одберете го третиот линк. Како одговор се добива порака од сервлетот `Servlet1`. Но, зошто се појавува порака од `Servlet1` кога ова барање го обработува `Servlet3`? Ако се погледне во кодот, може да се забележи дека навистина `Servlet3` го обработува барањето, но само парцијално бидејќи откако ќе ја инкрементира вредноста на атрибутот на бараќњето, истото го препраќа на доработка кај `Servlet1` кој потоа ја генерира следната порака:

-
- 1 `Servlet1`
 - 2 Parameters: `appSize = 100 configPages = 10 requestPage = 3`
 - 3 Attributes: `appCounter = 3 userCounter = 3 requestCounter = 2`
-

Состојбата на апликацијата откако по комплетно опслужување на барањето за адресата `"/s3"` е прикажано на сликата 2-20. Барањето иницијално го опслужува сервлетот `Servlet3` кој се иницијализира и се креира нишка за барањето. Методот

Вредноста на `configPages` е 10 бидејќи станува збор за параметар на сервлетот `Servlet1`, а вредноста на `requestPage` е 3 бидејќи при препраќањето на барањето во методот `forward()` во `Servlet3` беше наведена URL адресата `"/s1?page=3"`.

Вредноста на `appCounter` и `userCounter` се инкрементира за 1 и изнесува 3, додека пак вредноста на `requestCounter`, за разлика од претходно, сега изнесува 2. Причина за тоа е што барањето беше испратено најпрво до `Servlet3`, кој ја



Слика 2-20: Состојба на апликацијата откако ќе се повика "/s3"

постави вредноста на атрибутот на барањето на 1, а потоа тој проследи истото барање до **Servlet1** кој ја инкрементираше вредноста уште за 1.

Вратете се на главната страница и повторно кликнете на првиот линк. Се прикажува порака каде, во однос на првиот пат кога пристапивме до линкот, променети се само `appCounter` и `userCounter` бидејќи тие се менуваат со секое корисничко барање.

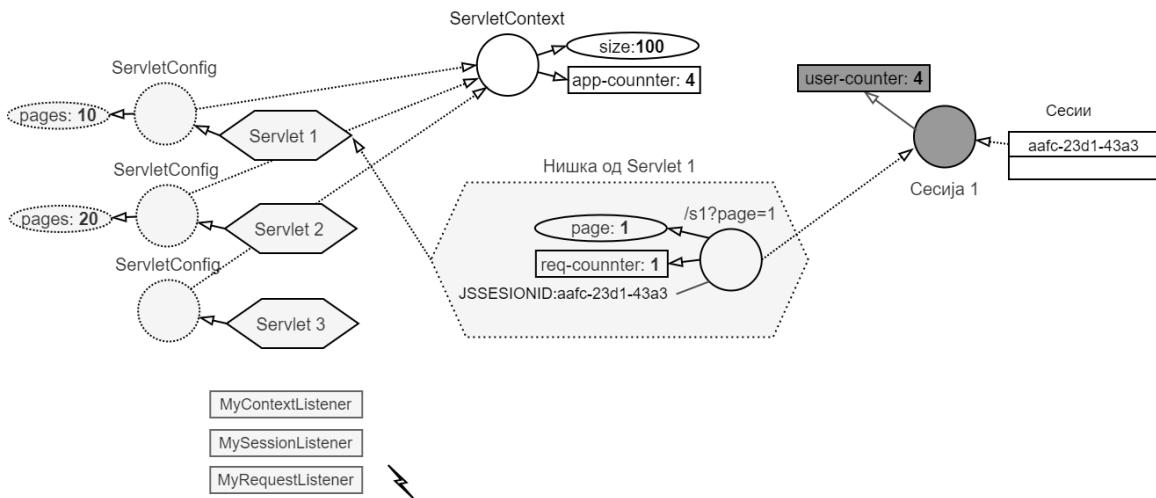
-
- 1 **Servlet1**
 - 2 Parameters: `appSize = 100 configPages = 10 requestPage = 1`
 - 3 Attributes: `appCounter = 4 userCounter = 4 requestCounter = 1`
-

Состојбата на апликацијата при повторно опслужување на барањето за адресата `"/s1?page=1"` е прикажано на сликата 2-21.

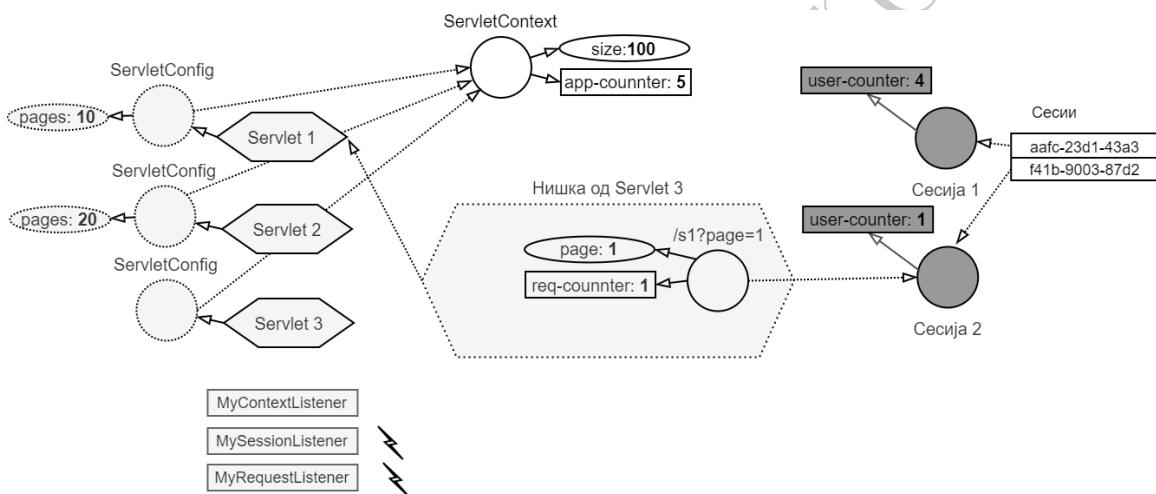
Отворете уште една инстанца на прелистувачот, но во анонимен начин на работа со цел да симулираме пристап до истата веб апликација, но од друг корисник. Внесете ја URL адресата на главната страница и притиснете на првиот линк. Како одговор се добива следната порака:

-
- 1 **Servlet1**
 - 2 Parameters: `appSize = 100 configPages = 10 requestPage = 1`
 - 3 Attributes: `appCounter = 5 userCounter = 1 requestCounter = 1`
-

На сликата 2-22 е прикажана состојбата на апликацијата откако ќе се опслужи барањето за адресата `"/s1?page=1"` во анонимен начин на работа. Вредноста на `appCounter` повторно се инкрементира во однос на претходната вредност, но вредноста на `userCounter` сега се поставува на 1. Зашто не се промени и таа



Слика 2-21: Состојба на апликацијата откако повторно ќе се повика `"/s1?page=1"`



Слика 2-22: Состојба на апликацијата откако повторно ќе се повика `"/s1?page=1"` од различен корисник

на вредност 5? Одговорот е во тоа што анонимната сесија не испраќа претходно снимени колачиња до серверот, па затоа новото барање не пренесува колаче според кое серверот ќе може да ја идентификува сесијата. Затоа, креира нова сесија, потоа се испалува настанот за креирање сесија кој го повикува методот `sessionCreated()` од слушачот `MySessionListener` и сесискиот атрибут `user-counter` за новата сесија ќе добие почетна вредност 0 која ќе се инкрементира за 1 со секое барање генерирано од анонимната сесија. Како што веќе напоменавме, колачето ќе биде зачувано само додека трае сесијата и ќе биде избришано откако ќе се затвори прелистувачот со анонимен начин на работа.

Ако се оди назад и ако се кликне на вториот линк, вредноста на `userCounter` ќе биде 2, а на `appCounter` ќе биде 6.

2.5 Филтри

Филтрите се уште една моќна компонента на J2EE која го поедноставува креирањето на веб апликации. Тие претставуваат модуларни компоненти управувани од контејнерот. Како и сервлетите, се вклучуваат во апликацијата преку конфигурациската датотека, но се целосно независни од нив, па затоа можат да се додаваат и отстрануваат од апликацијата без промена на нејзиниот код. Филтрите го пресретнуваат барањето пред да биде предадено на сервлет за обработка и овозможуваат негово претпроцесирање, но исто така го пресретнуваат и одговорот откако ќе биде обработен од сервлетот, пред да биде испратен на корисникот и на тој начин овозможуваат негово постпроцесирање. Всушност, контејнерот ги предава и објектот на барањето и објектот на одговорот на сервлетот, филтерот ги пресретнува и двата објекта и во двете насоки (кон сервлетот и од сервлетот), но во најголем број од случаите, пред објектите да бидат предадени на сервлетот важно е барањето, а кога сервлетот ќе заврши, важен е одговорт. Затоа, во текстот е наведено дека во едната насока се пресретнува барањето, а во другата одговорот иако во реалноста се предаваат двата објекта во двете насоки.

Зашто филтрите се од токлу голема важност во пракса? Замислете дека веб апликацијата треба да провери дали барањето доаѓа од автентициран корисник, односно дали содржи колаче со валиден идентификатор на сесија, пред да одлучи дали ќе го препрати на сервлет за обработка или ќе го пренасочи корисникот на страница за најава. Со досегашните алатки што ги имаме на располагање, ова би можело да се реши ако во кодот на секој сервлет се направи оваа проверка. Ваквото решение е неприфатливо во случај кога станува збор за покомплексна апликација со голем број на сервлети бидејќи истиот код треба да се имплементира кај секој сервлет поединечно. Уште повеќе, ако е потребно да се прават промени, тие би требало да се имплементираат кај сите сервлети. Ваквото решение е секако неодржливо и нескалабилно. Со помош на филтрите, кодот за автентикација се имплементира во филтер кој ќе ги пресретнува сите барања, без разлика кој сервлет треба да ги обработи, и доколу барањето содржи колаче со валиден сесиски идентификатор, барањето го проследува до дестинацискиот сервлет. Друг пример за употреба на филтрие е кога е потребно содржината на одговорот да се компресира со цел да се намали количината на податоци кои се испраќаат до клиентот. Во вакво сценарио, филтрите можат да го пресретнат одговорот од сервлетите и да го компресираат неговото тело пред да биде испратен до клиентот. Во пракса, филтрите најчесто се користат за имплементација

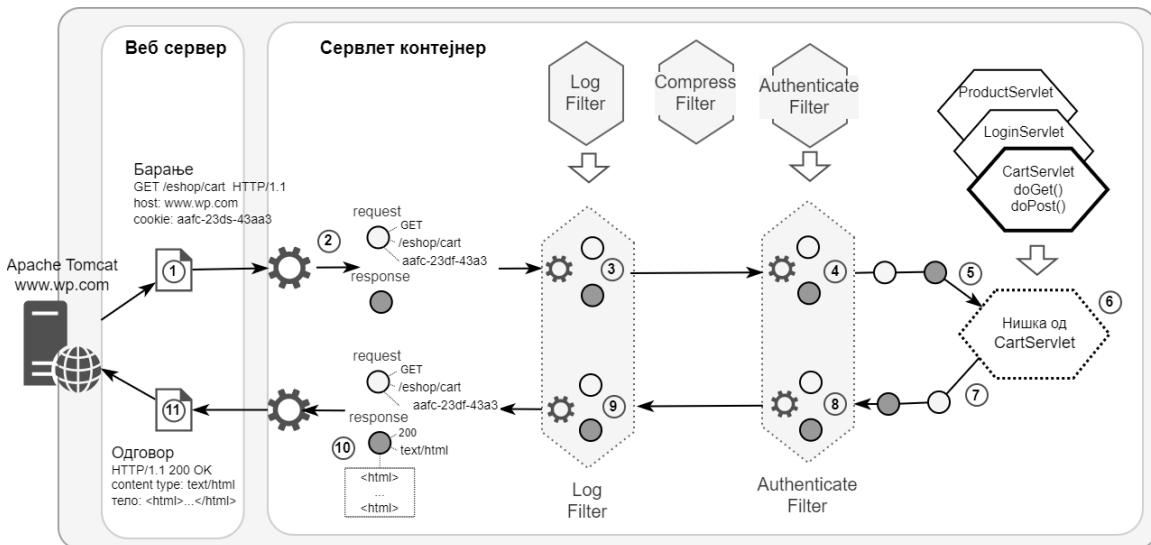
на автентикација, водење на дневник (анг. logging) за обработка на барања и одговори, валидација на влезни податоци, конверзија на податоци, компресија и декомпресија итн.

Една апликација може да содржи повеќе филтри за различни функционалности и сите тие може да се користат едно по друго во низа. Тоа значи дека, откако барањето ќе биде обработено од еден филтер, тоа се проследува на следниот филтер сè до крајот на низата кога барањето се предава на сервлетот. Редоследот на повикување на филтрите зависи од нивната конфигурација во `web.xml`. Откако сервлетот ќе го обработи барањето и ќе генерира одговор, тој се проследува до истите филтри, но во редослед обратен од редоследот на влезот во филтрите. Така, филтерот кој последен го обработил барањето ќе биде оној кој прв ќе го обработи одговорот и одговорот ќе се проследува до филтрите во насока кон филтерот кој прв го обработил влезното барање.

На сликата [2-23](#) е прикажан пример за целокупниот процес на обработка на барање и одговор од страна на веб контејнерот кога се применуваат филтри. Во примерот се дефинирани филтер за водење евидентија, компресија и автентификација, но само првиот и третиот филтер се пресликани за обработка на конкретната адреса на барањето.

1. Веб серверот му го предава барањето добиено од клиентот на сервлет контејнерот
2. Контејнерот ги креира објектите за барање и одговор
3. Контејнерот ги предава објектите на филтерот за логирање
4. Контејнерот ги предава објектите на филтерот за автентификација
5. Контејнерот ги предава објектите на сервлетот пресликан да го обработи барањето. Секој од филтрите може да ги спречи објектите да продолжат по описаната патека или пак да ги предаде на друг сервлет во зависност од имплементираната логика и содржината на барањето.
6. Сервлетот го обработува барањето и го пополнува одговорот со податоци,
7. Контејнерот ги проследува објектите до филтерот за автентификација
8. Контејнерот ги проследува објектите до филтерот за евидентија
9. Контејнерот генерира HTTP одговор од објектот за одговор
10. Контејнерот му го предава HTTP одговорот на веб серверо за достава до клиентот.

Освен за пресретнување на барањата кои стигнуваат директно од клиентот, филтрите може да се применат и на барањата препратени со методите `forward()` и `include()` на класата `RequestDispatcher`. Предефинираното однесување на контејнерот е да ги применува филтрите само на барањата од клиентите, но тоа однесување може да се конфигурира за секој филтер поединечно.



Слика 2-23: Претпроцесирање и постпроцесирање на барање и одговор со користење на филтри

2.5.1 Конфигурација на филтри

За да контејнерот знае за постоењето на филтри чија имплементација ја дефинираме во класи, и за да знае во кои случаи да ги применува во зависност од URL адресата на барањето, потребно е да се конфигурираат филтрите и нивните пресликувања во конфигурациската датотека `web.xml`.

Филтерот се дефинира помеѓу ознаките `<filter>...</filter>` како директно дете на коренот `<web-app>`. За секој филтер се дефинира

- `<filter-name>` со уникатно име за внатрешна употреба при преслукувањето и
- `<filter-class>` со име на имплементациска класа во елеменетот кое служи за контејнерот да ја лоцира класата при читање на филтерот.

Дополнително, во рамките на истиот јазол може да се дефинираат и иницијални параметри специфични за филтерот до кои може да се пристапи преку кодот на филтерот. Овие параметри служат само за читање и се дефинираат како парови од елементи `<param-name>`, `<param-value>` сместени помеѓу ознаките `<init-param>...<init-param>`.

Отако ќе се дефинираат филтрите, потребно е да се дефинира и описот на нивната употреба, односно, да се постават пресликувања на URL адреси во соодветен филтер. За разлика од сервлетите каде пресликувањето се прави само преку URL адреса, филтрите може да се пресликуваат и во еден или повеќе сервлети, што значи дека филтерот ќе биде повикан секогаш кога барањето ќе биде

наменето за еден од сервлетите наведени во пресликувањето. Секое пресликување се дефинира помеѓу ознаките `<filter-mapping>...</filter-mapping>` во кои се сместуваат елементите

- `<filter-name>` со името на претходно дефиниран филтер и
- `<url-pattern>` со URL адресата на барањето кое ќе го активира или
- `<servlet-name>` со името на сервлетот за кој е наменето барањето. Овој елемент може да содржи листа од повеќе сервлети одвоени со запирка.

За секое пресликување може дополнително да се дефинира за каков тип на барања ќе се однесува во зависност од тоа дали тие потекнуваат директно од клиентот или пак се проследени од `RequestDispatcher`. За таа цел, во зависност од потребата, се додава еден или повеќе елементи `<dispatcher>` во кои се сместува вредноста

- REQUEST за барања директно од клиентот
- FORWARD за барања кои потекнуваат од методот `forward()`
- INCLUDE за барања кои потекнуваат од методот `include()` или
- ERROR за барања кои се проследуваат до конфигурирана страница за грешки (ако таква страна не е конфигурирана во `web.xml`, филтерот не се активира).

Во кодниот сегмент 2.21 е даден пример во кој се дефинира филтерот `timer` имплементиран во класата `TimerFilter.class` со параметар `timeout` со вредност 2000, филтерот `authenticator` имплементиран во класата `AuthenticationFilter.class` и филтерот `cartLogger` имплементиран во класата `CartLoggerFilter.class`.

Филтерот `timer` ќе се активира за барањата кои потекнуваат директно од клиентите и оние проследени од некој сервлет со методот `forward()`, а филтерите `authenticator` и `cartLogger` ќе се активираат само за барањата кои потекнуваат директно од клиентите (иако експлицитно не е наведен елемент `<dispatcher>REQUEST</dispatcher>`, тоа е предефинирано однесување). Тоа значи дека ако има барање од клиентот за сервлетот `cart`, тогаш тоа ќе биде пресретнато најпрво од филтерот `timer`, потоа од `authenticator` и `cartLogger`. На крајот, барањето ќе биде доделено на сервлетот `cart`, кој откако ќе го генерира одговорот, истиот ќе биде пресретнат од филтерот `cartLogger`, па потоа од `authenticator` и на крајот од `timer` за потоа да биде испратен на клиентот.

```

1 <web-app>
2 ...
3 <filter>

```

```
4      <filter-name>timer</filter-name>
5      <filter-class>mk.ukim.finki.TimerFilter</filter-class>
6      <init-param>
7          <param-name>timeout</param-name>
8          <param-value>2000</param-value>
9      </init-param>
10     </filter>
11     <filter>
12         <filter-name>authenticator</filter-name>
13         <filter-class>mk.ukim.finki.AuthenticationFilter</filter-class>
14         <init-param>
15             <param-name>ignore-path</param-name>
16             <param-value>/guest</param-value>
17         </init-param>
18     </filter>
19     <filter>
20         <filter-name>cartLogger</filter-name>
21         <filter-class>mk.ukim.finki.CartLoggerFilter</filter-class>
22     </filter>
23     <filter-mapping>
24         <filter-name>timer</filter-name>
25         <url-pattern>/*</url-pattern>
26         <dispatcher>REQUEST</dispatcher>
27         <dispatcher>FORWARD</dispatcher>
28     </filter-mapping>
29     <filter-mapping>
30         <filter-name>authenticator</filter-name>
31         <servlet-name>/*</servlet-name>
32     </filter-mapping>
33     <filter-mapping>
34         <filter-name>cartLogger</filter-name>
35         <servlet-name>cart</servlet-name>
36     </filter-mapping>
37     ...
38 </web-app>
```

Изворен код 2.21: Конфигурација на филтри и нивно пресликување во `web.xml`

За разлика од пресликувањата кај сервлетите, каде што барањето се доделува на оној сервлет чија мапирана URL адреса најмногу се совпаѓа со неговата URL адресата, кај филтрите, барањето ќе биде доделено на сите филтри кои имаат совпаѓање со неговата URL адреса и тоа во редослед според кој се наведе.

дени пресликувањата во `web.xml`. Како што се изминуваат пресликувањата по редослед, најпрво се парсираат оние кои содржат елемент `<url-pattern>`. За секое совпаѓање филтерот се додава во веригата на филтри кои ќе го обработуваат барањето. Потоа, пресликувањата се изминуваат од почеток, но овој пат се парсираат само оние кои содржат елемент `<servlet-name>`.

Во кодниот исечок 2.22 е даден пример каде се дефинирани пресликувања на сервлети и филтри. Целта на примерот е да ја идентификуваме веригата на филтри кои ќе се активираат ако пристигне барање за URL адресата `"/cart/actions/add"`. При првото изминување на низата на пресликувања ќе се разгледуваат само пресликувањата со елемент елемент `<url-pattern>`. Од трите можни пресликувања F3, F4 и F5, совпаѓање има кај F3 и F5 бидејќи `"/cart/actions/*"` и `"/cart/*"` ја опфаќаат пристигнатата URL адреса `"/cart/actions/add"`. Во второто изминување на низата, се разгледуваат само пресликувањата со елемент `<url-pattern>`. Од двете можни пресликувања F1 и F2, совпаѓање има само кај F1 бидејќи овој филтер треба да се активира кога барањето ќе биде пресликано во сервлетот S1, кој пак се повикува кога адресата на барањето е `"/cart/actions/add"`. Според тоа, филтрите ќе бидат додадени во веригата од филтри F3, F5 и F1 кои ќе го пресретнат барањето и ќе го обработат пред да биде доделено на соодветниот сервлет.

```

1 <web-app>
2 ...
3   <servlet-mapping>
4     <servlet-name>S1</servlet-name>
5     <url-pattern>/cart/actions/add</url-pattern>
6   </servlet-mapping>
7   <servlet-mapping>
8     <servlet-name>S2</servlet-name>
9     <url-pattern>/cart/actions/edit</url-pattern>
10  </servlet-mapping>
11 ...
12  <filter-mapping>
13    <filter-name>F1</filter-name>
14    <servlet-name>S1</servlet-name>
15  </filter-mapping>
16  <filter-mapping>
17    <filter-name>F2</filter-name>
18    <servlet-name>S2</servlet-name>
19  </filter-mapping>
20  <filter-mapping>

```

```

21   <filter-name>F3</filter-name>
22   <url-pattern>/cart/actions/*</url-pattern>
23 </filter-mapping>
24 <filter-mapping>
25   <filter-name>F4</filter-name>
26   <url-pattern>/cart/actions/delete</url-pattern>
27 </filter-mapping>
28 <filter-mapping>
29   <filter-name>F5</filter-name>
30   <url-pattern>/cart/*</url-pattern>
31 </filter-mapping>
32 ...
33 </web-app>

```

Изворен код 2.22: Пресликување на повеќе филтри во web.xml

Освен во конфигурацискиата датотека, филтрите, пресликувањата и параметрите можат да се дефинираат во нивната имплементациската класа со помош на анотацијата `@WebFilter` во која се дефинира името на филтерот со параметарот `filterName`, мапираната URL адреса со параметарот `urlPattern`, пресликачите сервлети со параметарот `dispatcherNames`, типот на барања со параметарот `dispatcherTypes` и иницијалните параметри преку `initParams`.

Во продолжение е даден пример за конфигурација со анотацијата `@WebFilter` на филтерот `timer` од кодниот исечок 2.21. Анотацијата се употребува во самата класа за негова имплементација `timerFilter`, која за да може да биде филтер, мора да го имплементира интерфејсот `Filter`.

```

1 @WebFilter(
2   filterName = "timer",
3   urlPatterns = "/*",
4   dispatcherTypes = {DispatcherType.REQUEST, DispatcherType.FORWARD}
5   initParams = {
6     @WebInitParam(name = "timeout", value = "2000")
7   })
8 public class TimerFilter extends Filter {
9 ...
10 }

```

2.5.2 Имплементација на филтри

За да се имплементира филтер, потребно е да се дефинира класа која ќе го имплементира интерфејсот `Filter` кој ги дефинира следните методи:

```

1 public void init(FilterConfig config)
2 public void doFilter(ServletRequest request, ServletResponse response,
   FilterChain chain)
3 public void destroy()

```

Исто како и сервлетите, филтрите имаат животен циклус кој започнува со иницијализација од страна на контејнерот и повик на методот `init()` кој може да се искористи за да се иницијализираат ресурси во рамки на филтерот. Како аргумент на методот се пренесува објект од класата `FilterConfig` кој содржи референци кон параметрите дефинирани за филтерот во конфигурациската датотека (аналогно на `ServletConfig` кај сервлетите) и референца кон `ServletContext` за пристап до контекстот на апликацијата. Пред филтерот да терминира, се повикува методот `destroy()`.

За разлика од овие два метода кои се повикуваат само еднаш во животниот век на апликацијата, методот `doFilter()` се повикува при секоја активација на филтерот. Ја содржи целата логика на филтерот потребна за претпроцесирање и постпроцесирање на барањето и одговорот чии референци ги добива како влезни аргументи. Важно е да се забележи дека овие објекти не се од типот `HttpServletRequest` и `HttpServletResponse`, туку од генеричкиот тип `ServletRequest` и `ServletResponse` бидејќи филтрите можат да се користат и за апликации кои не се веб базирани и стриктно поврзани со клиентот преку протоколот HTTP. Сепак, за да се добие пристап до елементите специфични за HTTP барањето и одговорот, во имплементацијата на методот се прави едноставна конверзија (анг. casting).

Третиот аргумент на методот `doFilter()` е објект од типот `FilterChain` кој е од суштинско значење за проследување на барањето и одговорот. Тој ја содржи целата верига од филтри кои треба да се повикаат, па затоа се користи за да се активира следниот филтер во веригата. Ако станува збор за последниот филтер во веригата, тогаш објектите се пренесуваат на следниот ресурс, што во многу од случаите претставува сервлет. Оваа верига преставува низа од филтри која ја креира контејнерот и во неа ги сместува само оние филтри чие пресликување се совпаѓа со URL адресата на барањето. Веригата е дел од нишката која го обработува барањето и истата се брише пред одговорот да се испрати на клиентот. Меѓутоа, не се бришат самите филтри, туку низата од филтри со референци кон нив.

Интерфејсот `javaFilterChain` содржи само еден метод со следниот потпис:

```
public void doFilter(ServletRequest request, ServletResponse response)
```

Методот ги прима генеричките објекти на барањето и одговорот како аргументи и неговиот повик ги проследува до следниот филтер во веригата. Ако повикот се изостави, тогаш објектите нема да стигнат ниту до следните филтри од веригата ниту до дестинацискиот ресурс (сервлетот). Важно е да се направи разлика помеѓу овој метод `doFilter()` и методот `doFilter()` од интерфејсот `Filter`! Првиот само ги проследува објектите до следниот филтер од веригата и се повикува внатре, во имплементацијата на вториот, кој пак се повикува при активација на филтерот.

Генералната структура на методот `doFilter()` од интерфејсот `Filter` е следната:

```
1 public void doFilter(ServletRequest request, ServletResponse response)
2     throws ServletException, IOException {
3     //Preprocessing: the code above is executed before the objects are
4     //passed to the next filter in the chain in direction of activation
4     //towards the servlet
5     chain.doFilter(request, response);
6     //Postprocessing: the code below is executed when the objects are passed
6     //back in direction from the servlet towards the first filter
7 }
```

Сите линии на код до повикот на `chain.doFilter()` се извршуваат во фазата на претпроцесирање на објектите во моментот на активација на филтерот. Самиот метод ги проследува објектите до следниот ресурс во насока кон дестинацискиот ресурс (сервлетот). Линиите кои следат по повикот на методот се извршуваат во фазата на постпроцесирање, дури откако дестинацискиот ресурс ќе заврши со обработка на објектите и ќе почне да ги проследува во обратна насока на веригата, па затоа ќе се извршат откако претходниот филтер од веригата во насока кон клиентот ќе ги комплетира последните линии код од неговиот `doFilter()` метод.

Во кодниот исечок 2.23 е даден пример на имплементација на филтерот `timer` чија конфигурација е дадена во кодниот исечок 2.21. Целта на филтерот е да го измери вкупното време на процесирање на барањето од моментот кога контејнерот ќе му ги додели објектите на барање и одговор на филтерот, кој е и прв во низата филтри, сè до моментот кога овие објекти ќе му бидат вратени назад на контејнерот за да ги испрати до клиентот, што ќе се случи веднаш штом завршат последните линии код од филтерот бидејќи е последен филтер кој го обработува одговорот. На крајот, филтерот испишува порака во дневникот на контејнерот (на пример Tomcat) која исто така е видлива и на конзолата на користената IDE. Пораката ја содржи URL адресата на барањето и вкупното време на обработка.

Во случај тоа време да е подолго од параметарот на филтерот со име `timeout`, во пораката се допишува коментар дека обработката трае предолго време.

```

1 public class TimerFilter implements Filter {
2     private ServletContext context;
3     private long timeout;
4     public void init(FilterConfig config) throws ServletException {
5         this.context = config.getServletContext();
6         this.timeout = (long)config.getParameter("timeout");
7     }
8     public void destroy() {}
9     public void doFilter(ServletRequest request, ServletResponse response,
10                        FilterChain chain) throws ServletException {
11         long startTime = System.currentTimeMillis();
12         chain.doFilter(request, response);
13         long endTime = System.currentTimeMillis();
14         long total = endTime - startTime;
15         HttpServletRequest req = (HttpServletRequest) request;
16         String name = req.getRequestURI();
17         String comment = total > this.timeout ? "(too long!)": "";
18         this.context.log(name + " took " + total + " ms" + comment);
19     }
}

```

Изворен код 2.23: Имплементација на филтер за мерење на време на комплетна обработка на барање

За да се пристапи до контекстот и параметрите на филтерот се користи конфигурацискиот објект `FilterConfig` во методот `init()`. Со оглед на тоа што овој филтер е пресликан да се активира за сите URL адреси и тоа пресликување е наведено прво во `web.xml`, овој филтер ќе биде првиот филтер кој ќе ги пресретнува сите барања. Откако ќе се активира, во методот `doFilter` се чува тековниот моментот (изразен во ms од 1.1.1970), а потоа објектите на барањето и одговорот се проследуваат на следниот филтер во веригата. Откако објектите ќе стигнат до дестинацискиот сервлет и ќе ги поминат сите филтри од веригата во обратна насока, се стигнува до кодот во филтерот `timer` (кој е и последен) после повикот на `chain.doFilter()`. Се зема тековниот момент кој се користи за да се пресмета вкупното време на обработка на барањето, се прави споредба и се испишува соодветната порака во дневникот. За да се добие URL адресата на барањето, која е специфична за објекти од типот `HttpServletRequest`, потребно е да се направи конверзија од генеричкиот тип на барање `ServletRequest`.

Во кодниот сегмент 2.24 е дадена имплементацијата и на филтерот authenticator дефиниран во конфигурацијата во изворниот код 2.21. Главната функција на овој филтер е да проверди дали корисникот е автентициран. Ако е автентициран и или пак пристапува до адреса за која е дозволен и неавтентициран присатп, тогаш барањето се проследува до следниот ресурс во веригата на филтри. Во спротивно, барањето не се проследува понатаму во веригата (не се повикува методот `chain.doFilter()`) и наместо тоа, се проследува до страницата за најава.

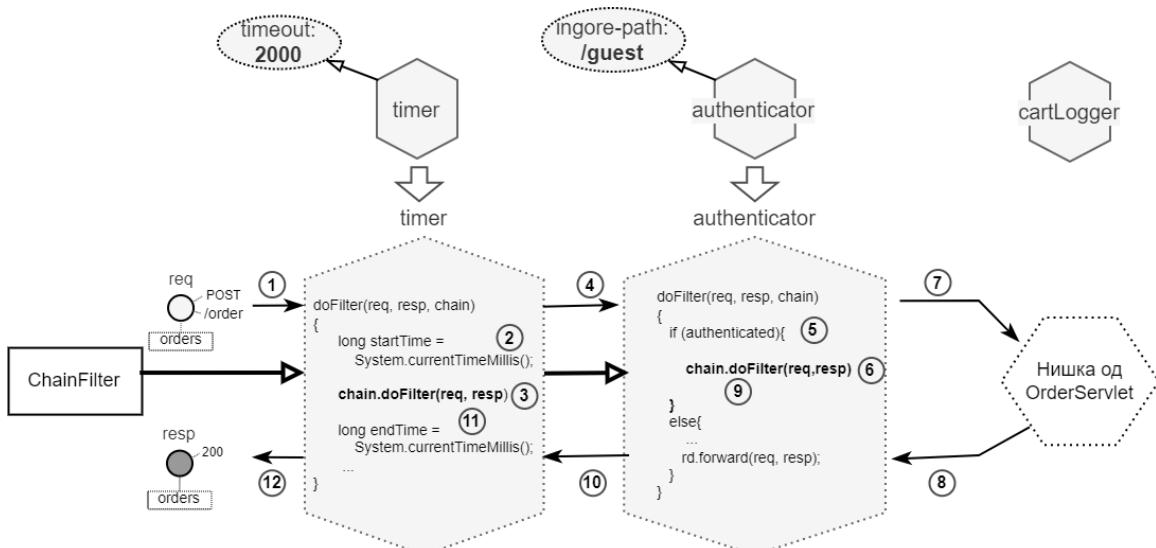
```

1 public class AuthenticationFilter implements Filter {
2     private String ignorePath;
3     public void init(FilterConfig config) throws ServletException {
4         this.ignorePath = config.getParameter("ignore-path");
5     }
6     public void destroy() {}
7     public void doFilter(ServletRequest request, ServletResponse response,
8             FilterChain chain) throws ServletException {
9         HttpServletRequest req = (HttpServletRequest) request;
10        if (req.getRequestURI().startsWith(ignorePath) || req.getUserPrincipal()
11            != null)
12            chain.doFilter(req, resp)
13        else{
14            RequestDispatcher rd=req.getRequestDispatcher("login.html");
15            rd.forward(req, resp);
16        }
17    }
18 }
```

Изворен код 2.24: Имплементација на филтер за автентикација

На сликата 2-24 се дадени чекорите на обработка на барањето и редоследот на извршување на различните секции од кодот на методот `doFilter` кaj дадените филтри при обработка на POST барање со патека `"/order"` кое потекнува од корисник кој веќе е автентициран. Иако во конфигурацијата `web.xml` е дефиниран и филтерот `cartLogger`, тој нема да стане дел од веригата на филтри `filterChain` бидејќи патеката на барањето не се совпаѓа со пресликувањето дефинирано за овој филтер. Чекорите се следни:

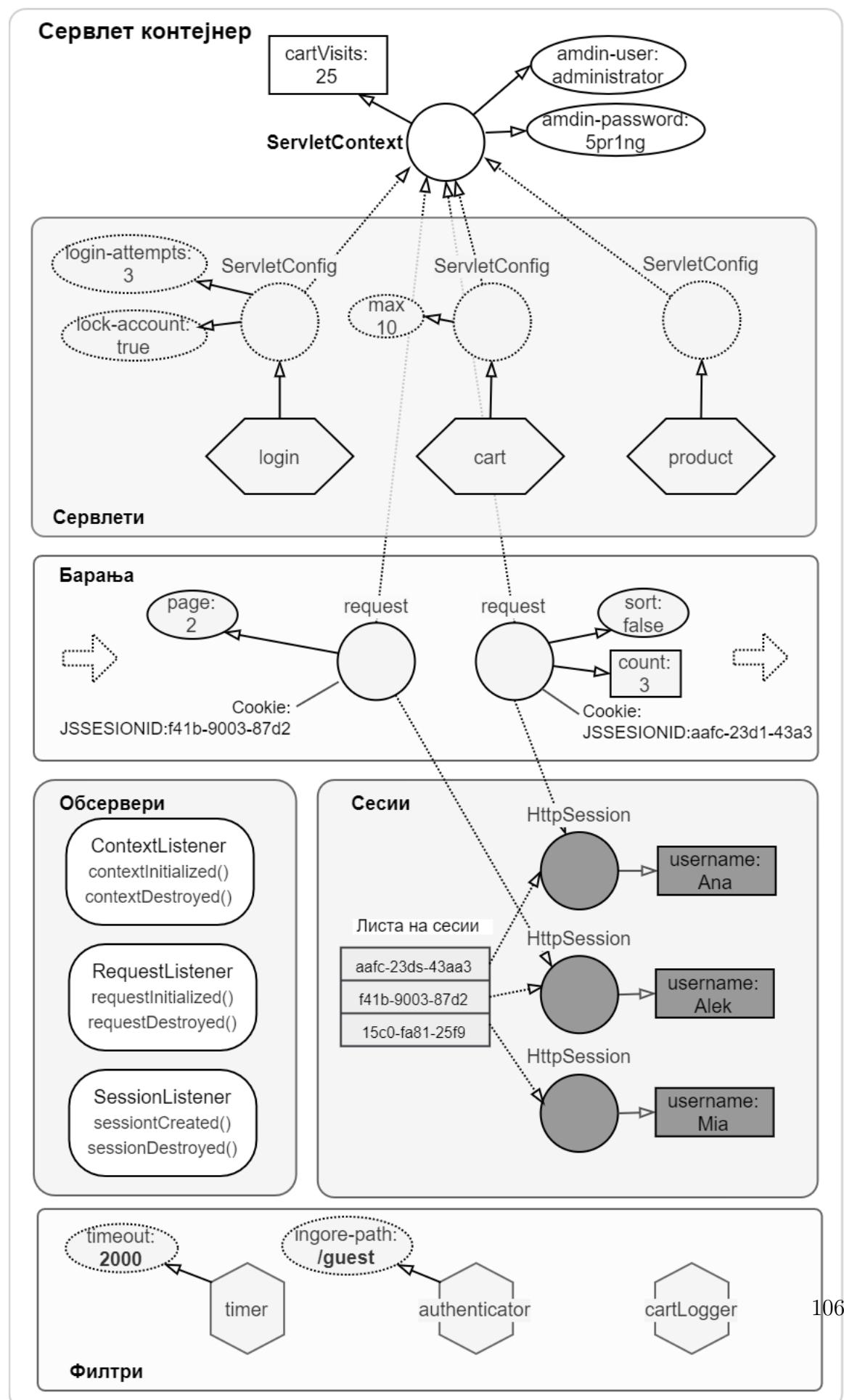
1. Барањето доаѓа до контејнерот кој според неговата адреса креира верига од филтри `filterChain` составена од филтрите `timer` и `authenticator` чиј редослед го определува според редоследотот на пресликувањата во `web.xml` кои се совпаѓаат со адресата



Слика 2-24: Редослед на извршување на код од филтри при обработка на барање

2. Се повикува методот `doFilter()` на првиот филтер од веригата `timer`
3. Се извршува наредбата `chain.doFilter()` која го предава барањето на следниот филтер од веригата
4. Барањето се предава на филтерот `authenticator`
5. Се извршува методот `doFilter()` на филтерот
6. Бидејќи станува збор за автентициран корисник, се извршува наредбата `chain.doFilter()` за проследување на барањето на следниот ресурс
7. Бидејќи `authenticator` е последниот филтер од веригата, барањето се предава на сервлетот `OrderServlet` кој е пресликан да го обработи барањето
8. Сервлетот го уредува одговорот и го враќа назад во веригата од филтри
9. Се извршуваат линиите код после `chain.doFilter()` на филтерот `authenticator`. Бидејќи во примерот нема други наредби, одговорот се проследува до следниот филтер во насока кон клиентот
10. Одговорот се предава на филтерот `timer`
11. Се извршуваат линиите код после `chain.doFilter()` на филтерот `timer` во кои се пресметува вкупното траење на целокупната обработка и се печати во дневникот на контејнерот
12. Одговорот се предава на контејнерот за да биде испратен на веб серверот и клиентот, а потоа контејнерот ги брише објектите на барањето и одговорот, нишката од сервлетот и веригата на филтри `chainFilter`

По воведувањето на филтрите може да се добие комплетна слика за организација на управуваните компоненти. Слика 2-25 претставува пропишување на слика 2-16 каде се воведени филтрите и параметрите кои се дефинирани за нив.



Слика 2-25: Комплетен преглед на управувани компоненти, параметри и атрибути

Глава 3

Запознавање со Spring рамката

3.1 Spring рамка

Spring рамката е дизајнирана со цел да го олесни креирањето на комплексни Јава апликации. Рамката е со отворен код (open source, англ.) и е поддржана од голема и активна заедница, која обезбедува континуирана поддршка за различни реални сценарија. Токму оваа заедница помага Spring рамката континуирано да еволуира и да ги поддржува потребите на реалните апликации.

Од верзијата 5.1 на Spring рамката, потребно е користење на JDK 8+ (Java SE 8+). Оваа рамка поддржува различни сценарија за поставување (deployment scenarios, англ.):

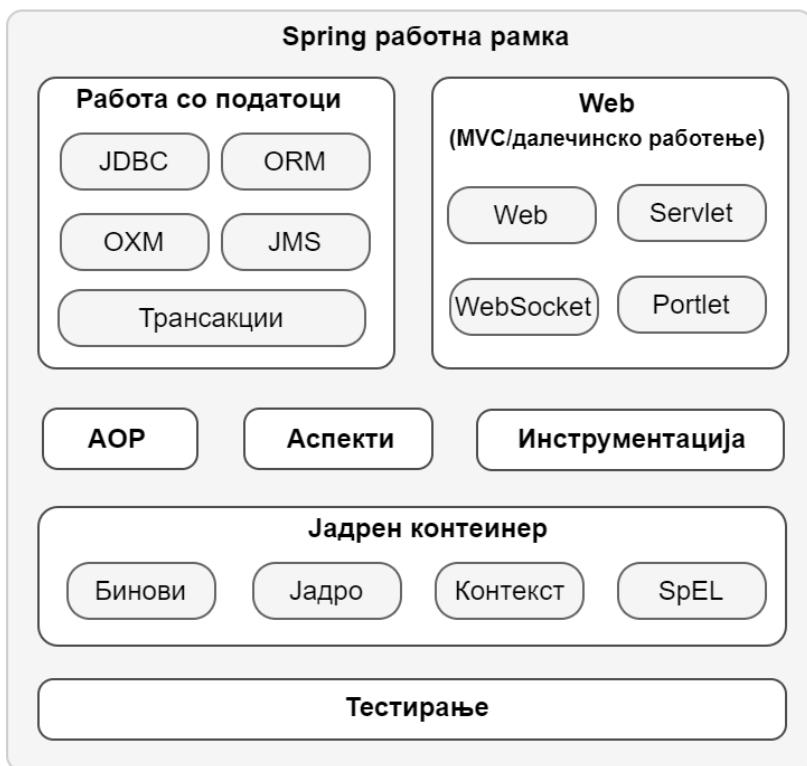
- Стартување на апликации во рамки на веб апликациски сервер кој е независен од апликацијата (пр. поставување во Apache Tomcat);
- Стартување на апликацијата со вгнезден (embedded, англ.) сервер;
- Стартување на апликација без користење на сервер (executable jar, англ.).

Spring рамката се состои од модулите прикажани на слика 3-1. Овие модули соработуваат и се надоградуваат меѓусебно. Сите модули од слика 3-1 претставуваат jar-датотеки¹ што може да се вклучат како зависности во рамките на проектот, доколку нѝ треба одредена технологија. Во табела 3.1 се прикажани дел од основните модули од Spring рамката кои ќе бидат обработени или споменати во рамките на оваа книга. Повеќето од модулите имаат зависност од некој друг модул во Spring рамката. Основниот модул е исклучок од ова правило. Слика 3-2 дава преглед на најчесто користените модули и нивните зависности од другите модули.

¹JAR (Java Archive) датотеки се архиви кои содржат компајлирани Јава класи, ресурси и други спакувани елементи. Се користат за групирање и дистрибуција на Јава апликации и библиотеки.

Име	Артефакт	Опис
Spring Core	spring-core	Основниот модул на Spring кој ги обезбедува основните функционалности и инверзија на контрола.
Spring Beans	spring-beans	Модул за управување со бинови (beans) во Spring контекстот.
Spring AOP	spring-aop	Модул за аспектно-ориентирано програмирање (Aspect-Oriented Programming - AOP) во Spring апликации.
Spring Expression Language	spring-expression	Модул за користење на изрази (expressions) во Spring контекстот и анотациите.
Spring Context	spring-context	Модул кој обезбедува напредни функционалности за управување со Spring контекстот и компонентите.
Spring Web	spring-web	Модул за развој на веб апликации кој ги обезбедува функционалностите за модел-поглед-контролер архитектура.
Spring WebMVC	spring-webmvc	Spring MVC пакет за поддршка за употреба во Servlet околнота.
Spring Tx	spring-tx	Модул за трансакциско управување во Spring апликации. Обезбедува анотации и конфигурација за управување на трансакции.
Spring JDBC	spring-jdbc	Модул за употреба на JDBC (Java Database Connectivity) за комуникација со релациони бази на податоци во Spring апликации.
Spring ORM	spring-orm	Модул кој го обезбедува интегрирањето на ORM (Object-Relational Mapping) технологии како Hibernate, JPA и други во Spring апликации.
Spring Data	spring-data	Модул за упростено пристапување до податоци со користење на Spring. Обезбедува абстракции и готови решенија за работа со различни типови на податоци и бази на податоци.
Spring Thymeleaf	spring-boot-starter-thymeleaf	Модул за интеграција на Thymeleaf, модерен шаблонски мотор, во Spring апликации. Овозможува развој на динамички HTML страници и шаблони.

Табела 3.1: Модули од Spring рамката



Слика 3-1: Модули од Spring рамката

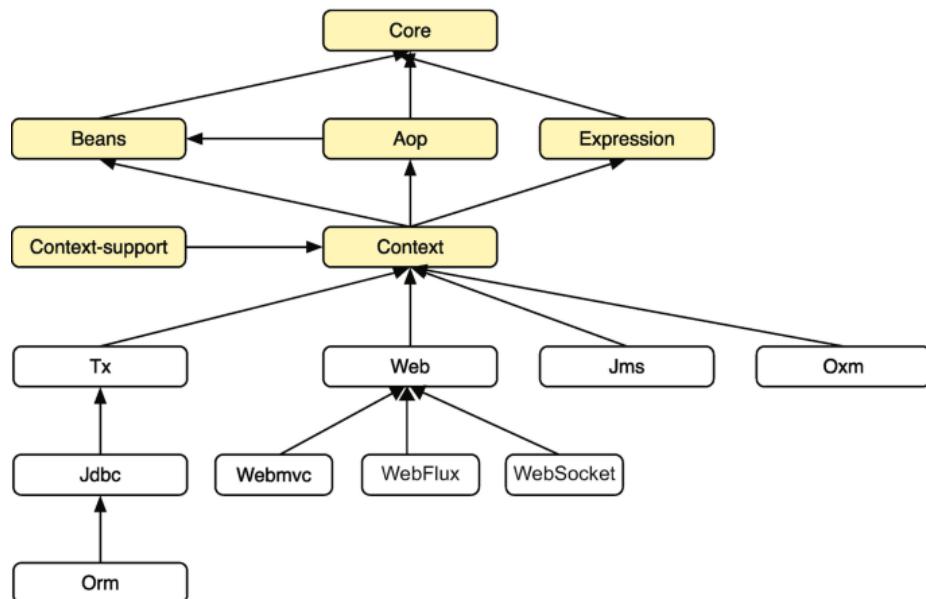
Во рамките на оваа книга, основните функционалности на библиотеките Spring Core, Beans и Context ќе бидат разработени во рамките на ова, воведно поглавје. Деталите за Spring Web и WebMVC ќе бидат разработени во поглавјете 4, Spring Tx, Jdbc и Orm ќе бидат разработени во 5, а Spring Expression и security ќе бидат разработени во поглавјето 6. Дел од функционалностите на останатите библиотеки од Spring рамката ќе бидат само споменати доколку има потреба, но нема да се обработат во целост, за да може да се задржи фокусот на веб апликациите.

3.2 Дизајн филозофија на Spring

Кога се изучува нова рамка за развој, покрај тоа што треба да научиме како се користи, треба да ги разбереме и дизајн принципите кои таа ги следи (design philosophy, англ.).

Водечките принципи на Spring рамката се:

- **Избор на секое ниво.** Spring рамката овозможува да се одложат дизајн одлуките колку што е можно подоцна. На пример, Spring овозможува да се



Слика 3-2: Зависности на модулите од Spring рамката

промени начинот на зачувување на податоците само преку конфигурација, без притоа да се промени кодот.

- **Поддршка на различни перспективи.** Spring рамката овозможува флексибилност и не ѝ ограничува во начините на работа. Поддржува широк опсег на апликациски потреби со различни перспективи. На пример, во однос на конфигурацијата се поддржува XML конфигурација, конфигурација со анотации, конфигурација со својства, конфигурациски класи, како и комбинирање на сите претходно споменати опции.
- **Одржување на компатибилност наназад.** Една од главните причини за рас пространетоста на Spring рамката е грижата за апликациите кои се веќе изработени да продолжат да функционираат и во поновите верзии. Ова е клучно затоа што апликациите се наменети да адресираат реални бизнис сценарија, кои постоеле и пред изработка на веб апликација и кои треба да ја надживеат технологијата. Еволуцијата на Spring рамката е управувана внимателно, со цел бројот на некомпатибилности помеѓу верзиите да се сведе до минимум.
- **Грижа за дизајнот на интерфејсите (API).** Се води сметка дизајнирите интерфејси да бидат интуитивни и да опстојат што е можно подолго и низ што е можно повеќе верзии. Се користи пристап на именување на интерфејсите според намената (анг. Intention driven interface), со кој се обезбедува лесно читлив код кој имплицитно ја дефинира причината за постоење на интерфејсот. На овој начин се намалува потребата од документација и се зголемува читливоста на кодот.

- **Поставување на високи стандарди за квалитет на код.** Се обрнува особено внимание на процесот на креирање и одржување на тековна и концизна документација. Spring рамката е еден од ретките проекти кој може да се пофали со чист код и структура, без циркуларни зависности помеѓу пакетите.

3.2.1 SOLID принципи

SOLID принципите претставуваат множество од пет основни принципи во објектно-ориентираното програмирање кои служат за креирање флексибилни, модуларни софтверски системи кои се лесни за одржување. Овие принципи се користат насекаде низ Spring рамката и ја подржуваат дизајн филозофијата на рамката. Овие принципи претставуваат добри практики кои помагаат да се креира код што е податочно-ориентиран, со добра структура и лесно проширлив.

Основните принципи на SOLID се во продолжение.

Принцип на единствена одговорност (Single Responsibility Principle (SRP), англ.)

Секоја класа или модул треба да има само една одговорност т.е. само една причина за промена. Ова значи дека секоја класа треба да биде одговорна само за една конкретна работа или функционалност. Имањето на една одговорност овозможува полесно тестирање на кодот, затоа што треба да се проверат помалку функционалности и се потребни помалку зависности за иницијализација на тест сценаријата. Ако класата има повеќе од една одговорност, тогаш станува комплицирана и тешка за одржување.

```

1 class PaymentService {
2     boolean badPay(Double val, String acc) {
3         if(isCasys(acc)) { // requested and implemented in 2018
4             // do payment with casysLib
5             // ~ 50 lines
6         } else if (isPaypal(acc)) { // requested and implemented in 2020
7             // do payment with paypalLib
8             // ~
9         } else if (isStripe(acc)) { // requested in 2023
10            // work in progress: payment with stripeLib
11            // ~
12        } else {
13            // unsupported account exception

```

```
14    }
15  }
16 }
```

Изворен код 3.1: Пример за класа која не го почитува принципот на единствена одговорност

Во изворниот код 3.1 е даден пример за класа која не го почитува принципот на единствена одговорност. Класата `PaymentService` има повеќе од една одговорност, бидејќи се грижи за плаќање со различни платежни системи. Преку овој пример ќе објасниме зошто е важно да се почитува овој принцип од гледна точка на тестирањето и одржувањето на кодот.

Како што се гледа од коментарите на кодот, оваа класа иницијално е развиена за да овозможи плаќање преку Casys системот за плаќање во 2018 година, по што во 2020 е проширена да подржи плаќање со Raurnal, за да на крај во 2023 се побара да подржува плаќање преку Stripe системот. Во овој пример ќе се претпостави дека системите за плаќање имаат свои библиотеки кои го олеснуваат развојот и кои се додадени како зависности на проектот. Дополнително, за да се обезбеди точност и функционалност на кодот, потребно е да се дефинираат тестови.

Со тоа што не се следи принципот за единствена одговорност, креирањето на тестови за `PaymentService` класата станува многу комплицирано. За да се тестира функционалноста за плаќање со Casys системот, потребно е да се иницијализираат зависностите за Casys системот, а исто така и да се иницијализираат зависностите за Paypal и Stripe системите, бидејќи се наоѓаат во истиот код. Ова значи дека тестовите за Casys системот ќе зависат и од зависностите за Paypal и Stripe системите, што не е потребно и не е практично. Исто така, тестовите за Paypal и Stripe системите ќе зависат и од зависностите за Casys системот, што е непотребно и не е практично.

Дополнително, во 2023 година е потребно да се додаде поддршка за плаќање со Stripe системот. За да се додаде поддршка за плаќање со Stripe системот, потребно е да се додаде нова зависност за Stripe системот, да се додаде нова логика за плаќање со Stripe системот и да се додадат тестови за плаќање со Stripe системот. Ова значи дека во 2023 година, кога ќе се додаде поддршка за плаќање со Stripe системот, тестовите за плаќање со Casys и Raurnal системите ќе треба да се променат и да се додадат зависностите за Stripe системот, што не е потребно и не е практично. Исто така, тестовите за плаќање со Stripe системот ќе зависат и од зависностите за Casys и Raurnal системите, што не е потребно и не е практично.

За да се следи принципот за единствена одговорност, оваа класа треба да се промени во три класи, секоја со посебна одговорност за плаќање со соодветен

платежен систем. Меѓутоа, прво ќе се запознаеме со следните 2 принципи, по што ќе објасниме како ќе изгледа кодот кој ги следи сите нив.

Принцип на отвореност за проширување -затвореност за модификација (Open-Closed Principle (OCP), англ.)

Овој принцип дефинира дека кодот треба да биде отворен за проширување, но затворен за модификација. Овој принцип дефинира дека нема да се менува кодот кој што е функционален и коректен при додавање на нови функционалности, т.е. **тоа што работи, не се менува**. Функционалниот код подразбира дека е проверен преку тестови, па негово менување за додавање на нови функционалности значи дека треба да се направи менување и во тестовите.

Ова значи дека треба да постои можност за додавање на нова функционалност без да се менува постоечкиот код. Проширувањето треба да се врши преку додавање нови класи, наследување и преоптоварување на методи, а не преку модификација на тековниот код. Ова овозможува менување на тековна или додавање нова функционалност без да се менува постоечкиот код, а со тоа и потенцијално да се генерираат непосакувани странични ефекти.

Класата `PaymentService`, која е прикажана во [3.1](#) не го следи ни принципот отвореност за проширување, затвореност за модификација. Веќе видовме дека додавањето на нови функционалности во оваа класа ќе предизвика промена на постоечкиот код и тестови, што е индикација дека овој принцип не е следен во дизајнот на класата.

Лисков принцип за замена (Liskov Substitution Principle (LSP), англ.)

Објектите од родител-класата треба да имаат можност да бидат заменети со објекти од нивните наследни класи, без да се наруши функционалноста на програмата. Ова значи дека секоја наследена класа треба да ги задржи сите основни карактеристики и функционалности на својата родител-класа. Принципот го формулира Барбара Лисков и го изразува следново: "Ако S е подтип на T, тогаш објектите од T може да бидат заменети со објекти од S без да се наруши правилната функционалност на програмата".

Со други зборови, доколку имаме класа T и од неа наследува класа S, секој објект од класата T може да биде заменет со објект од класата S без да се промени очекуваната функционалност на програмата. Ова означува дека сите методи што ги имплементира S треба да ги задоволат истите контракти и услови како и методите од T.

Овој принцип има неколку импликации и предности:

- Повеќекратна употребливост и флексибилност: Според LSP, кодот кој ја користи класата T може да работи со објекти од класата S, што го зголемува степенот на употребливост и флексибилност на системот.
- Соодветно користење на наследувањето: Принципот ги насочува развиваите да користат наследување само кога има јасна врска на типот помеѓу класите, што помага во подобрување на дизајнот и читливоста на кодот.
- Идентификација на неправилности во моделот: Кога не се почитува LSP, тоа може да открие неправилности во моделот, како што се недоволна апстракција или лош дизајн на врските при наследување.

Со почитување на LSP, добиваме модели кои се флексибилни и лесни за користење, каде што можеме да замениме објекти со нивни подтипови без да ја нарушуваме функционалноста на програмата.

```

1 interface PaymentProcessor {
2     boolean pay(Double val, String acc);
3     boolean accept(String acc);
4 }
5 class PaymentService {
6     public List<PaymentProcessor> processors=Stream.of(
7         new CasysPaymentProcessor(),
8         new PaypalPaymentProcessor(),
9         new StripePaymentProcessor()
10    ).collect(Collectors.toList());
11
12    boolean pay(Double val, String acc) {
13        for(PaymentProcessor p: processors) {
14            if(p.accept(acc)) {
15                p.pay(val, acc);
16                return true;
17            }
18        }
19        return false;
20    }
21 }
```

Изворен код 3.2: Пример за класа која го почитува принципот на единствена одговорност и отвореност за проширување преку почитување на принципот на замена на Лисков

Според дефинициите на овој принцип, во насока на надминување на недостатоците на `PaymentService` класата од изворниот код [3.1](#), принципот за

замена на Лисков дефинира дека е препорачливо да се дефинира интерфејс `PaymentProcessor` и да се имплементираат класи `CasysPaymentProcessor`, `PaypalPaymentProcessor` и `StripePaymentProcessor` според неговата дефиниција. Изворниот код [3.2](#) прикажува пример за овој принцип. Како што се гледа во овој пример, класата `PaymentService` сега има само една одговорност, а тоа е да ги процесира плаќањата. Со тоа што класата `PaymentService` има само една одговорност, тестовите за неа се поедноставни и лесни за креирање. Исто така, класата `PaymentService` сега е отворена за проширување, бидејќи може да се додадат нови класи кои ќе ги имплементираат интерфејсот `PaymentProcessor` и да се додадат во листата `processors` без да се менува постоечкиот код. Исто така, класата `PaymentService` сега е затворена за модификација, бидејќи не е потребно да се менува постоечкиот код за да се додаде нова функционалност, туку само да се додаде нова класа која ќе го имплементира интерфејсот `PaymentProcessor` и да се додаде во листата `processors`. Исто така, класата `PaymentService` сега го почитува и принципот за замена на Лисков, бидејќи сите објекти од класата `PaymentService` може да бидат заменети со објекти од класите `CasysPaymentProcessor`, `PaypalPaymentProcessor` и `StripePaymentProcessor` без да се наруши функционалноста на програмата.

Принцип на разграничување на интерфејси (Interface Segregation Principle (ISP), англ.)

Овој принцип налага класите да не зависат од интерфејси кои не ги користат целосно. Односно, интерфејсите треба да бидат специфични за потребите на класите кои ги користат, т.е. да содржат само методи кои се потребни за конкретната функционалност. Овој принцип ја поддржува идејата за разграничување на интерфејсите на помалку специфични делови, така што класите ќе ги користат само оние делови од интерфејсот што ги употребуваат.

Според принципот за разграничување на интерфејсите, една класа или модул не би требало да зависи од методи или функционалности кои не ги користи. Во вакви случаи, би требало да се разграничват интерфејсите и да се креираат поспецифични интерфејси за конкретни потреби. Ова помага во постигнувањето на следните предnosti:

- Разграниченост и чистота на интерфејсите: Интерфејсите стануваат поедноставни и лесни за разбирање, бидејќи содржат само методи кои се потребни за конкретна функционалност.
- Модуларност и флексибилност: Промената на еден дел од системот не треба да предизвика промена на други делови кои не зависат од тој дел.

- Повторна употреба на код: Интерфејсите можат повторно да се искористат и многу лесно да се имплементираат во различни контексти и од страна на различни клиентски класи.
- Полесно тестирање на кодот: со користење на поспецифични интерфејси, тестовите се полесни за креирање затоа што е потребно симулирање на помалку зависности, т.е. нема потреба до симулација на однесувањето на зависностите кои не се директно потребни за функционалностите на клиентските класи.

Пример за примена на ISP би било разграничување на голем интерфејс за управување со кориснички профили во повеќе специфични интерфејси, како што се интерфејс за регистрација, интерфејс за најава, интерфејс за управување со лозинки, итн. Клиентските класи потоа можат да ги користат само интерфејсите што ги употребуваат, без да зависат од непотребните функционалности.

Принцип на инверзија на зависност (Dependency Inversion Principle (DIP), англ.)

Модулите треба да зависат од апстракции, а не од конкретни имплементации. Ова значи дека треба да користиме интерфејси или апстрактни класи за комуникација помеѓу модулите, наместо да зависиме директно од конкретни класи. Овој принцип подетално ќе го објасниме подоцна во оваа глава.

```

1 class PaymentService {
2     public List<PaymentProcessor> processors=new ArrayList<>();
3     // omitted code, same as in the previous example
4 }
5
6 class PaymentApplication {
7     public static void main(String[] args) {
8         PaymentService payService = new PaymentService();
9         payService.processors.addAll(
10             Stream.of(new CasysPaymentProcessor(),
11                     new PaypalPaymentProcessor(),
12                     new StripePaymentProcessor()
13             ).collect(Collectors.toList()));
14     }
15 }
```

Изворен код 3.3: Пример за класа која ги почитува SOLID принципите

Доколку се анализира изворниот код [3.2](#), може да се забележи дека овде не се почитува принципот за инверзија на контролата. Класата `PaymentService` зависи од конкретни имплементации на интерфејсот `PaymentProcessor`, а не од апстракција. Ова значи дека ако сакаме да додадеме нова имплементација на интерфејсот `PaymentProcessor`, треба да се менува постоечкиот код, со што се нарушува принципот за затвореност за модификација. Исто така, ако сакаме да ја тестираме класата `PaymentService`, повторно треба да се иницијализираат сите зависности за сите имплементации на интерфејсот `PaymentProcessor`, што не е добра практика. Решение на овој проблем може да се понуди со користење на принципот за инверзија на контролата, прикажан во изворниот код [3.3](#). Во овој пример, класата `PaymentService` зависи од апстракцијата `PaymentProcessor`, а не од конкретни имплементации. Ова овозможува да се додаде нова имплементација на интерфејсот `PaymentProcessor` без да се менува постоечкиот код, што значи дека класата `PaymentService` е затворена за модификација. Исто така, ова овозможува да се тестира класата `PaymentService` без да се иницијализираат зависностите за сите имплементации на интерфејсот `PaymentProcessor`, што значи дека класата `PaymentService` е отворена за проширување. Во овој пример, контролата за тоа кои зависности се префрли од класата `PaymentService` во класата `PaymentApplication`, т.е. контролата за тоа кои зависности се вметнуваат во класата `PaymentService` е инвертирана и самата класа `PaymentService` не е повеќе свесна за имплементациите за плаќањето кои ги користи.

Преку овие примери покажавме како се почитуваат принципите на единствена одговорност, отвореност за проширување, затвореност за модификација, замена на Лисков, разграничување на интерфејсите и инверзија на зависностите. Како што може да се забележи, со следење на овие принципи, кодот е почист, полесен за читање, полесен за тестирање и најважно од се, полесен за одржување.

3.3 Вметнување зависности

Во оваа секција ќе биде објаснето како функционира концептот на вметнување на зависности (dependencies injection, англ.), што се зависности (dependencies, англ.) на дадена класа, како Spring дознава за тие зависности и како ги вметнува соодветните зависности во рамките на самата класа.

```
1 public class ProductServiceImpl implements ProductService {  
2     private final ProductRepository productRepository;  
3     protected CategoryRepository categoryRepository;  
4  
5     public static ProductService create(ProductRepository productRepository,  
6                                         CategoryRepository categoryRepository) {  
7         return new ProductServiceImpl(productRepository, categoryRepository);  
8     }  
9 }
```

```

6           CategoryRepository categoryRepository) {
7               ProductServiceImpl instance = new
8                   ProductServiceImpl(productRepository, categoryRepository);
9               // do some further initialization or configuration
10              return instance;
11         }
12
13     public ProductServiceImpl(ProductRepository productRepository,
14                             CategoryRepository categoryRepository) {
15         this.productRepository = productRepository;
16         this.categoryRepository = categoryRepository;
17     }
18
19     public void setCategoryRepository(CategoryRepository categoryRepository)
20     {
21         this.categoryRepository = categoryRepository;
22     }
23
24     // the rest of the code is omitted
25 }
```

Изворен код 3.4: ProductServiceImpl

Во класата `ProductServiceImpl` дадена во изворникот код 3.4, се дефинирани две својства: `productRepository` и `categoryRepository`. Овие својства се од тип `ProductRepository` и `CategoryRepository`, соодветно. Со оваа дефиниција се кажува дека `ProductServiceImpl` зависи од типовите `CategoryRepository` и од `ProductRepository`.

Она кое што се поставува како прашање е како ќе се вметнат овие две зависности. Од аспект на Јава програмскиот јазик, има неколку можности да се постави дадено својство:

- **Вметнување преку конструктор (constructor injection, англ.):** Првата можност е вметнување на вредностите преку конструктор, каде што вредностите на зависностите се поставуваат преку самиот конструктор.
- **Вметнување преку поставување на својство (property injection, англ.):** Вториот начин е вредноста да се постави директно на својството. Но, за да може да се постави вредноста, треба својството да е видливо (`protected` или `public`) во контекстот на поставувањето, или пак со користење на рефлексија (Java reflection, англ.)².

²Рефлексијата во Јава програмскиот јазик овозможува интерфејс за прелистување и манипулација со својствата и методите на класите.

- **Вметнување преку метод за поставување вредност (setter injection, англ.):** Третиот начин е да се искористи метод за поставување на вредност (setter, англ.) за да се проследи вредноста на зависноста.
- **Вметнување преку метод за креирање инстанца (factory method injection, англ.):** Последниот начин е да се искористи метод за креирање на инстанца (factory method, англ.), преку кој ќе се креира објектот и ќе му се вметнат потребните зависности. Овој начин на креирање и вметнување на зависностите е соодветен кога покрај креирањето, потребно е да се направат дополнителни прилагодувања на објектот или неговите зависности.

За да покажеме како функционира вметнувањето на зависностите и каде вистински има примена, ќе дефинираме еден тест. Во тестот ќе се покаже што всушност значи креирање на `ProductServiceImpl` класата и ќе се покажат четирите начини на вметнување на зависноста.

```

1  public class ProductServiceImplTest {
2
3      ProductServiceImpl instance;
4
5      @Before
6      public void init() {
7          // instantiate dependencies
8          ProductRepository nr= Mockito.mock(ProductRepository.class);
9          CategoryRepository cr=Mockito.mock(CategoryRepository.class);
10
11         // constructor injection
12         instance = new ProductServiceImpl(nr, cr);
13
14         // property injection
15         instance.categoryRepository = cr;
16
17         // setter injection
18         instance.setCategoryRepository(cr);
19
20         // factory method injection
21         ProductService factoryInstance = ProductServiceImpl.create(nr, cr);
22
23     }
24 }
```

Изворен код 3.5: `ProductServiceImplTest`

Првиот дел од тестот прикажан во изворниот код [3.5](#), го содржи делот со креирање на зависностите. Во овој пример се користи библиотека Mockito, која што всушност нѝ креира „симулирана“ имплементација на `ProductRepository` и `CategoryRepository` интерфејсите.

Зашто библиотеки за симулирани имплементации во тестовите?

Се добива поедноставување на самиот тест, бидејќи не се користат вистинските имплементации на зависностите. Во нашиот случај, имплементацијата на `ProductRepository` би значела поврзување со базата на податоци. Ова би значело дека тестирањето на кодот не е можно без поврзување со база на податоци, т.е. без инсталiran и стартуван систем за базата на податоци во тест околината, многу често има потреба од иницијализација на базата на податоци со соодветни вредности, што всушност го отежнува самото тестирање. Во овој конкретен случај, потребно е да се провери дали логиката напишана во `ProductServiceImpl` е точна, што не зависи од податоците кои ќе се добијат од база на податоци или од друга локација. Единствено е потребна имплементација на `ProductRepository` која ќе се однесува согласно спецификацијата на интерфејсот. Токму затоа, наместо да се користат вистинските имплементации, се користат „симулирани“ (mock, англ.) имплементации, со што се олеснува самото тестирање.

По креирањето на зависностите, во примерот се прикажани и трите начини за вметнување на зависностите. Меѓутоа, кодот кој го погледнавме беше дизајниран да поддржува вметнување на зависностите, а со тоа и негово поедноставно тестирање и замена на имплементацијата на репозиториумите кои се користат како зависност. Сепак, за да овозможиме вметнување на зависностите, потребно е објектот да не ги специфицира зависностите во самата имплементација, како во примерот од изворниот код [3.6](#). Во ваквите случаи се вели дека има закодирани (hard coded, англ.) зависности кои не може да се променат во време на извршување на кодот. Поради тоа, доколку треба да се тестира класата `HardCodedDependencyProductServiceImpl`, не би можело да искористи симулирана верзија од `ProductRepository` и би морало да се иницијализира базата на податоци пред извршување на самиот тест, што всушност значи дека не може да напишеме Unit тест за оваа класа.

```

1 public class HardCodedDependencyProductServiceImpl implements
2   ProductService {
3     private final ProductRepository productRepository;
4     protected CategoryRepository categoryRepository;
```

```

4
5   public HardCodedDependencyProductServiceImpl(CategoryRepository
6     categoryRepository) {
7     this.productRepository = new DbProductRepositoryImpl("hardcoded db
8       connection");
9     this.categoryRepository = categoryRepository;
10    }

```

Изворен код 3.6: HardCodedDependencyProductServiceImpl

3.4 Инверзија на контролата со користење на Spring рамката

Зошто е потребна инверзија на контролата? До сега објаснивме што се тоа зависности и што значи да вметнеме дадени зависности на дадена класа. Во овој процес ние моравме самите да ги креираме имплементациите на секоја од зависностите и да ги поставиме на новите објекти кои ги креираме. Во примерот со тестот имавме само 2 зависности кои треба да се креираат. Меѓутоа, во реални апликации често имаме десетици, па и стотици објекти кои треба да ги креираме и да им ги вметнеме зависностите. Креирањето на објектите и поврзувањето на зависностите во реални проекти може да биде долготраен и здодевен процес. За среќа, тука е Spring рамката која го поедноставува овој процес со инверзија на контролата.

Принципот на инверзија на контролата овозможува декларативно означување на зависностите и нивно имплицитно поврзување во процесот на креирање на објектите. Секоја инстанца од класа која е зачувана во контекстот се нарекува „гравче“ (bean, англ.) во Spring терминологијата. Во рамките на оваа книга, ние ќе го користиме терминот „бин“, наместо преводот „гравче“. Биновите се објекти кои се креираат, составуваат и управуваат од страна на Spring контејнерот за инвертирање на контролата (Spring inversion of control - Spring IoC, англ.). Биновите и нивните зависности се управуваат во однос од конфигурираните мета-податоци на Spring IoC контејнерот.

Пакетите `org.springframework.beans` и `org.springframework.context` ги содржат основните дефиниции на Spring IoC контејнерот. Интерфејсот `BeanFactory` ги дефинира напредните механизми за конфигурација, кои овозможуваат управување со сите типови на објекти. `ApplicationContext` е интерфејс кој е изведен од `BeanFactory` и ги додава следните својства:

- Полесна интеграција со функционалностите за аспект-ориентирано програмирање (AOP) од Spring рамката;
- Справување со ресурси за интернационализација;
- Објавување на настани;
- Воведува поддршка за специфични контексти, како што е `WebApplicationContext`, кој се користи кај веб апликациите.

Накратко, `BeanFactory` ги дефинира механизмите за конфигурација и основните функционалности на Spring рамката, додека пак `ApplicationContext` додава специфични функционалности кои се потребни за бизнис апликациите.

3.4.1 Регистрирање на бинови

Spring контејнерот е тој што најпрво треба да ги пронајде компонентите и да креира бинови од нив. Двата најчести начини кои се користат за да ги регистрираме биновите во Spring апликацискиот контекст се преку XML конфигурација и преку Јава конфигурација со скенирање на анатациите на компонентите (`component scan`, англ.). XML конфигурацијата експлицитно ги наведува класите кои ќе бидат вклучени во апликацискиот контекст и тута имаме можност експлицитно да ги наведеме зависностите кои ќе бидат вметнати на секоја од компонентите.

Во кодот подолу е прикажана пример XML конфигурација.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://www.springframework.org/schema/beans
5          https://www.springframework.org/schema/beans/spring-beans.xsd">
6      <!-- scan the content of a package-->
7      <component-scan package="mk.ukim.finki.wp.eshop.repository"/>
8      <!-- services -->
9      <bean id="productService"
10         class="mk.ukim.finki.wp.eshop.service.ProductServiceImpl">
11         <constructor-arg index="0" ref="productRepository"/>
12         <constructor-arg index="1" ref="categoryRepository"/>
13     </bean>
14
15     <!-- enable spring annotation processing for DI -->
16     <context:annotation-config/>
17
18     <!-- more bean definitions can go here -->
19

```

 20 </beans>

Изворен код 3.7: Spring XML конфигурација

Во оваа конфигурација, експлицитно ги наведуваме биновите кои сакаме да бидат креирани, заедно со зависностите кои сакаме да им ги вметнеме. Сепак, Spring е флексибilen и овозможува автоматско вметнување на зависностите на својствата аnotирани со `@Autowired`. Оваа анотација дава инструкција на контејнерот да го пронајде најсоодветниот бин и да го вметне како вредност на аnotираното свойство. За да се овозможи процесирањето на овие анотации потребно е да се додаде `<context:annotation-config/>` во XML конфигурацијата. Но, дури и со ова поедноставување, сепак конфигурацијата на големи апликации би била комплицирана.

```

1 @Configuration
2 @ComponentScan("mk.ukim.finki.wp.eshop.repository")
3 public class AppConfig {
4
5     // If not defined explicitly,
6     // the bean id is the name of the method
7     @Bean
8     public ProductService productService(
9             ProductRepository productRepository,
10            CategoryRepository categoryRepository) {
11        return new ProductServiceImpl(productRepository, categoryRepository);
12    }
13 }
```

Изворен код 3.8: Јава базирана конфигурација на Spring IoC контејнерот

Покрај конфигурацијата на Spring IoC контејнерот со XML, постои можност и за Јава базирана конфигурација (изворен код 3.8). Во овој случај, наместо да користиме XML датотека со спецификација на биновите кои треба да се регистрираат во контејнерот, користиме Јава класи кои се аnotирани со `@Configuration` и кои содржат методи аnotирани со `@Bean`. Овие методи всушност одговараат на `<bean id="some-id" class="path.to.the.ComponentClass"/>` од XML конфигурацијата и се користат за да креираат инстанци кои ќе се регистрираат во контејнерот. Во примероците изворните кодови 3.7 и 3.8 е прикажана идентична конфигурација. Во кодот 3.9 се покажува како се креира Spring IoC контејнер во кој ќе се искористи Јава или XML конфигурација. Од вака креираниот контекст може да се повика некоја од варијантите на `getBean()` методот за да се добие регистриран бин според идентификатор или тип.

```

1  @Configuration
2  class AppConfig {
3      public static void main(String[] args) {
4          ApplicationContext javaCtx = new AnnotationConfigApplicationContext(
5              AppConfig.class);
6
7          ApplicationContext xmlCtx = new ClassPathXmlApplicationContext(
8              "app-config.xml");
9
10         // we can also use xmlCtx for finding a bean that implements
11         // the ProductService interface
12         ProductService produtService = javaCtx.getBean(ProductService.class);
13         // productService.doStuff();
14     }
15 }
```

Изворен код 3.9: Креирање на Spring IoC контејнр со XML и Јава конфигурација

Најголемото поедноставување на конфигурацијата на Spring IoC контејнерот се добива преку механизмот за скенирање на компоненти (component scan, англ.). Овој механизам може да го вклучиме на два основни начини: преку додавање на `<component-scan package="package.path"/>` или преку `@ComponentScan(package="package.path")` анатацијата³. Овој механизам му кажува на Spring IoC контејнерот дека треба да ги скенира сите класи во рамките на наведениот пакет и да ги пронајде Spring компонентите. Компоненти во контекст на конфигурацијата на контејнерот се сите класи кои што се анотираны со `@Component` или некоја од нејзините изведени анатации: `@ControllerAdvice`, `@Controller`, `@Repository`, `@JsonComponent`, `@TestComponent`, `@Service`, `@Configuration`, итн. Подоцна контејнерот креира бинови од компонентите кои се пронајдени.

Разлика помеѓу `@Component` и `@Bean`

`@Component` е анатација наменета за означување на класи за кои Spring IoC контејнерот имплицитно ќе креира и регистрира бинови. Овие анатации се процесираат во рамки на скенирањето на компонентите.

`@Bean` е анатација наменета за означување на методи чиј резултат ќе биде експлицитно креиран бин кој ќе се регистрира во рамките на IoC контејнерот.

³Постои и варијанта во која наместо пакет се наведува класа, но и во овој случај всушност се скенира пакетот во кој се наоѓа дадената класа.

3.4.2 Животен циклус на биновите

Важно е да се разбере дека Spring биновите се управувани објекти од страна на Spring контејнерот за инверзија на контролата. Во процесот на управување, Spring контејнерот ја превзема одговорноста за креирање и инстанцирање на биновите, вметнување на зависности, управување со животниот циклус и уништување на биновите. Разбирањето на животниот циклус на Spring биновите ви овозможува да го управувате овој процес и да ги контролирате фазите во кои се извршуваат одредени активности.

Процесот на управување на Spring биновите започнува со креирање на контејнерот (`ApplicationContext`) и регистрирање на биновите преку конфигурација, анотации или скенирање на класи. Кога бинот е потребен, контејнерот ја креира инстанцата на бинот и ги вметнува сите потребни зависности. Потоа се повикуваат посебни методи (иницијализација) за конфигурирање на бинот пред да биде достапен за користење.

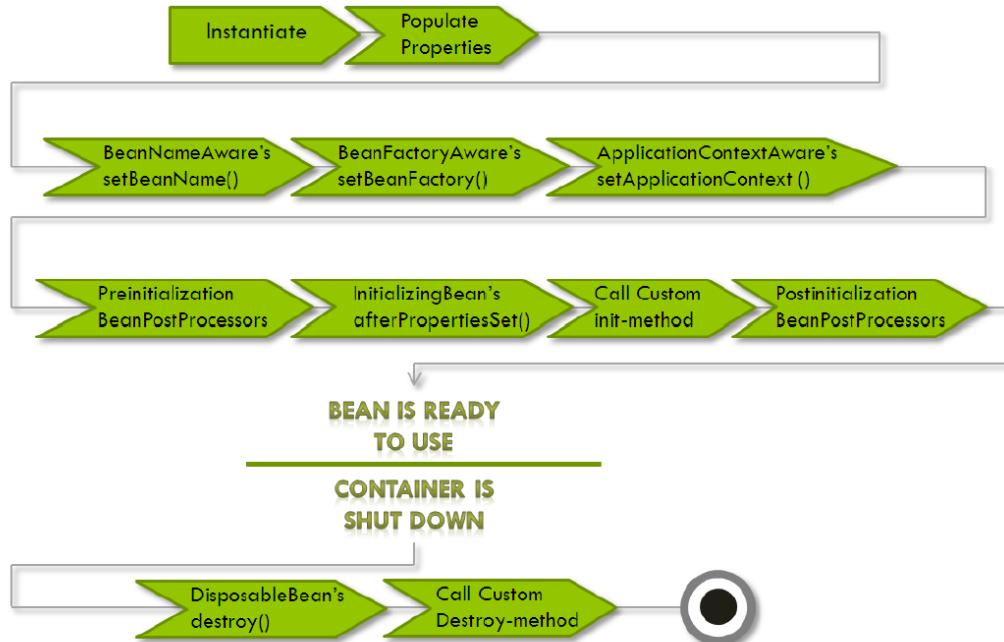
Кога бинот не е повеќе потребен, Spring контејнерот го управува неговото уништување. Според конфигурацијата, може да се повика посебен метод (деструкција) на бинот пред да биде уништен.

Разбирањето на овој процес е важно бидејќи ви овозможува да ги оптимизирате перформансите и ресурсите на апликацијата. Исто така, можете да ги примените посебните акции или бизнис-логика во различни фази од животниот циклус на бинот. Примери за ова вклучуваат поставување на предефинирани вредности, валидација, отварање и затворање на конекции кон бази на податоци, почетно пополнување на податоци, итн.

Во целост, разбирањето на животниот циклус на Spring биновите ви овозможува да ги максимизирате можностите и предностите на Spring рамката и да креирате и управувате со бинови кои работат ефикасно и правилно во рамките на вашата апликација.

Сликата [3-3](#) го прикажува животниот циклус на стартивање на типичен бин, додека е вчитан во контекст на апликацијата Spring. Како што е прикажано, контејнерот извршува неколку чекори за поставување, пред бинот да биде подготвен за употреба. Попрекизно, на сликата се прикажани следниве чекори:

- Spring го инстанцира бинот.
- Spring ги вметнува вредностите и референците на бинот во неговите својства.
- Ако бинот го имплементира интерфејсот `BeanNameAware`, Spring му ја предава идентификацијата на бинот на методата `setBeanName()`.
- Ако бинот го имплементира интерфејсот `BeanFactoryAware`, Spring ја повикува методата `setBeanFactory()`, пренесувајќи ја самата фабрика на бинови.



Слика 3-3: Животен циклус на Spring бинови

- Ако било кој од биновите го имплементира интерфејсот ApplicationContextAware, Spring ќе ја повика методата setApplicationContext(), пренесувајќи референца кон опкружувачкиот контекст на апликацијата.
- Ако некој од биновите го имплементира интерфејсот BeanPostProcessor, Spring ќе го повика неговиот метод postProcessBeforeInitialization().
- Ако некој од биновите го имплементира интерфејсот InitializingBean, Spring ќе го повика неговиот метод afterPropertiesSet(). Исто така, ако бинот е деклариран со init-method, тогаш ќе биде повикан соодветниот метод за иницијализација.
- Ако има бинови кои го имплементираат интерфејсот BeanPostProcessor, Spring ќе ги повика нивните методи postProcessAfterInitialization().
- Во овој момент, бинот е подготвен за употреба од страна на апликацијата и ќе остане во контекстот на апликацијата сè додека контекстот на апликацијата не биде уништен.
- Ако некој од биновите го имплементира интерфејсот DisposableBean, тогаш Spring ќе го повика неговиот метод destroy(). Исто така, ако бинот е деклариран со destroy-method, тогаш ќе биде повикан соодветниот метод за уништување.

3.4.3 Автоматско вметнување на зависности

Автоматското вметнување на зависности (autowiring, англ.) се користи во следните случаи:

- при вметнување на зависности анотирани со `@Autowired` анотацијата,
- при вметнување на аргументи на конструкторот за компоненти од кои се креираат бинови и
- при вметнување на аргументи на методите анотирани со `@Bean`.

Spring ги поддржува следните модели за автоматско вметнување на зависности:

- **no**: Не се користи автоматско вметнување на зависностите. Сите зависности мора експлицитно да се вметнат. Ова е предефиниранот модел.
- **byName**: Во овој модел се вметнуваат зависностите според името на бинот. Во овој случај, името на својството и името на бинот треба да бидат исти.
- **byType**: Во овој модел се вметнуваат зависностите според типот, при што е дозволено својството и бинот да имаат различни имиња.
- **constructor**: Во овој модел се вметнуваат зависностите со повик на конструкторот на класата:
 - Доколку има само еден конструктор, ќе се искористи тој конструктор⁴.
 - Ако има само еден конструктор анотиран со `@Autowired`, ќе се искористи токму тој.
 - Ако има повеќе конструктори, од кои ниту еден не е анотиран со `@Autowired`, ќе се повика предефиниранот конструктор без аргументи. Ако не постои таков конструктор, ќе се фрли грешката `BeanCreationException`.
 - Ако има повеќе конструктори анотирани со `@Autowired(required=false)`, ќе се повика конструкторот со најмногу аргументи чии зависности може да се задоволат.
 - Доколку имаме повеќе конструктори анотирани со `@Autowired(required=false)` и еден или повеќе конструктори анотирани со `@Autowired(required=true)` ќе биде фрлена грешката `BeanCreationException`. Ова сценарио не е дозволено.
- **autodetect**: Во овој модел, прво се прави обид за вметнување на зависностите со `constructor` моделот. Доколку не успее, тогаш се прави обид со `byType` моделот.

⁴Во Spring, почнувајќи од верзија 4.3, ако дадена класа има само еден конструктор, кај тој конструктор не мора да ја користиме `@Autowired` анотацијата, а сепак тој конструктор ќе се користи за вметнување на зависностите преку него.

Меѓутоа, овие модели пред сè се користат во XML базираните конфигурации со својството `<bean id="" class="" autowire="byName"/>`. Доколку се користи Јава конфигурација со анотации, тогаш анотацијата `@Autowired` се разрешува според моделот `byType`. Во овој случај, доколку анотацијата е искористена на својство, се бара бин според типот на својството во Spring IoC контејнерот. Доколку анотацијата е искористена на метод или на конструктор, тогаш за секој од аргументите се бара бин во Spring IoC контејнерот за неговиот тип.

Предефинираното однесување на `@Autowired` анотацијата е `@Autowired(required=true)`, што значи дека доколку не е пронајден еднозначен кандидат за секоја зависност од методот, ќе биде фрлен исклучок и нема да може да се иницијализира Spring IoC контејнерот. Доколку сакаме да го избегнеме ова однесување, потребно е да се користи `@Autowired(required=false)`. Во ситуациите кога се користи `@Autowired` анотацијата на методи или конструктори со повеќе аргументи, а сакаме дел од овие аргументи да не бидат задолжителни, тоа може да го изведеме со користење на `Optional<T>` типот за соодветниот аргумент за верзиите $>=5.0$.

Доколку се користи `@Autowired Collection<X> all`, тогаш во колекцијата `all` ќе бидат вметнати сите регистрирани бинови во Spring IoC контејнерот од типот `X` (вклучително и од типовите кои го наследуваат `X`).

Доколку се користи `@Autowired Map<String, X> idToBean`, тогаш во мапата `idToBean` ќе бидат вметнати сите регистрирани бинови во Spring IoC контејнерот од типот `X` како вредности, а за клучевите ќе се искористат имињата на биновите, т.е. нивните идентификатори. Од оваа причина, типот на клучевите мора да биде `String`.

Последниот дел кој ќе го разгледаме од делот за автоматско вметнување на зависностите е процесот на разрешување на еднозначен кандидат за дадена зависност анотирана со `@Autowired`:

- Доколку има само еден регистриран бин во Spring IoC контејнерот од соодветниот тип, ќе се вметне тој бин како зависност на анотираното свойство (истото важи и за аргумент на метод или конструктор).
- Доколку нема ниту еден бин од тој тип во Spring IoC контејнерот, ќе се фрли исклучокот `NoSuchBeanDefinitionException`.
- Доколку има повеќе регистрирани бинови од типот на зависноста, а името на еден од тие бинови е исто со името на својството (или аргументот на методот), тогаш ќе се искористи соодветниот бин како зависност. На пример, доколку во Spring IoC контејнерот имаме 3 бинови од потребниот тип со имиња "x1" "x2" и "x3" а имаме потреба да ја разрешиме зависноста `@Autowired X x1`, тогаш ќе се искористи бинот со име "x1" од контекстот како

еднозначен кандидат.

- Доколку има повеќе регистрирани бинови од типот на зависноста, а името на ниту еден од тие бинови не е исто со името на својството (или аргументот на методот), тогаш ќе се фрли исклучокот `NoUniqueBeanDefinitionException`.
 - Во случаите кога го знаеме специфичниот бин кој сакаме да го вметнеме, но сакаме да искористиме друго име на променливата, се препорачува означување на зависноста со анотацијата `@Qualifier("beanId")`, преку која експлицитно дефинираме кој бин сакаме да се искористи како зависност. Во погорниот пример, зависноста би требало да биде анотирана на следниот начин: `@Autowired @Qualifier("x1") X myX`.
 - Од друга страна пак, кога бинот кој сакаме да го вметнеме зависи од околината во која ја стартуваме апликацијата, се препорачува користење на `@Profile("client-x1")` анотацијата при декларацијата на компонентата `@Profile("client-x1") @Component` или методот за регистрација на бин `@Profile("client-x1") @Bean`. На овој начин, соодветниот бин ќе биде регистриран во Spring IoC контекстот само доколку активниот профил е поставен на `"client-x1"` при стартирањето на апликацијата. Активниот профил во Spring Boot апликациите се поставува со својството `spring.profiles.active=dev,client-x1` во датотеката `application.properties`.

Зошто се препорачува вметнување на зависности преку конструктор?

- Јасно се идентификуваат зависностите.
- Не постои начин да се заборави зависност при тестирање или во која било друга околност на инстанцирање на објектот. Не би можело да се креира објектот без проследување на вредност за зависноста во конструкторот.
- Не е потребна рефлексија за да се постават зависностите. Може да се користат симулирани имплементации и експлицитно да се вметнат преку конструкторот за полесно тестирање.
- Со секоја нова зависност треба да се модифицираат тестовите за компонентата. Ова придонесува да имаме ажурирани тестови за имплементацијата.
- Голем број на зависности ги прави конструкторите тешки за користење. **Ова е индикација дека се нарушуваат single responsibility и interface segregation принципите од SOLID шаблонот.** Во вакви случаи е препорачливо иницијалната компонента да се подели во повеќе помали компоненти со јасни и специфични задачи.
- Се препорачува зависностите да бидат final, што помага за робусност и сигурност на нишките (thread-safety).

3.5 Опции за Конфигурирање во Spring

Овде ќе се прикажат опциите за дефинирање на животниот век на биновите преку `@Scope` анотацијата, конфигурирање на вредности во Spring рамката преку `@Value` анотацијата, начинот на поставување на вредностите во различни околини, како и механизмот за конфигурација на профили со `@Profile` анотацијата.

3.5.1 Животен век на биновите

Животниот век на биновите во Spring контејнерот за инверзија на контролата може да се контролира со помош на `@Scope` анотацијата. Оваа анотација дефинира кога ќе бидат креирани биновите, како и колку долго ќе трае нивниот животен век во рамки на контејнерот.

Оваа анотација може да се искористи на ниво на класа или на метод. Кога се искористува на ниво на класа за означување на животниот век на компонентите во комбинација со `@Component` или нејзините изведени анотации. Кога се искористува на ниво на метод е дополнување на `@Bean` анотацијата и го дефинира животниот век на бинот кој се креира со тој метод.

Постојат следните опции за животен век на биновите во Spring:

- `@Scope("singleton")`: Ова е стандардниот животен век на биновите во Spring. Кога се користи овој животен век, Spring контејнерот креира само една инстанца од бинот и ја користи за сите последователни барања за тој бин преку `ctx.getBean("singletonName")`. Ова значи дека сите барања за бинот ќе вратат иста инстанца. Ова е предефиниранот животен век на биновите во Spring. Биновите конфигурирани на овој начин се креираат при иницијализација на контејнерот за инверзија на контролата и остануваат во меморија се додека контејнерот не биде уништен (пр. при исклучување на апликацијата).
- `@Scope("prototype")`: Кога се користи овој животен век, Spring контејнерот креира нова инстанца од бинот за секое барање т.е. секое повик на `ctx.getBean("prototypeName")`. Биновите конфигурирани на овој начин **само се креираат при секое барање** и не се зачувуваат во контејнерот за инверзија на контролата. Ова е корисно кога бинот содржи податоци кои се разликуваат за секоја инстанца. Меѓутоа, биновите од типот `prototype` не се уништуваат автоматски од Spring контејнерот за инверзија на контрола, туку нивната меморија се ослободува со собирачот на губре (garbage collector) кога немаат повеќе референции кон нив. Ова означува дека ако бинот има алоцирано ресурси кои треба да ги ослободи, треба да се води сметка секогаш рачно да се повикаат методите за оваа намена, бидејќи по креирањето на бинот, Spring повеќе не се грижи за него.
- `@Scope("request")`: Кога се користи овој животен век, Spring контејнерот креира една инстанца од бинот за секој HTTP request. Биновите конфигурирани на овој начин се креираат при секој HTTP request и се уништуваат при завршување на истиот.
- `@Scope("session")`: Кога се користи овој животен век, Spring контејнерот креира една инстанца од бинот за секој HTTP session. Биновите конфигурирани на овој начин се креираат при секоја HTTP сесија и се уништуваат при завршување на истата.

3.5.2 Конфигурација на вредности во Spring

`@Value` анотацијата во Spring овозможува вметнување на вредности во биновите од различни извори. Се користи за вчитување вредности од property фајлови, системски променливи, аргументи на командната линија итн. Примерот во изворниот код [3.10](#) го прикажува користењето на оваа анотација.

¹ `@Configuration`

```

2  @PropertySource("classpath:application.properties")
3  public class AppConfig {
4
5      @Value("${db.url}")
6      private String dbUrl;
7
8      @Value("${db.username}")
9      private String dbUsername;
10     // ...
11 }

```

Изворен код 3.10: Вметнување на вредности со @Value анотацијата

@Value анотацијата во Spring е мокна алатка за вметнување на конфигурациски вредности во биновите. Овој механизам зголемува флексибилност за управување со апликацијата, што го прави идеален избор за конфигурација во Spring проекти.

3.5.3 Надворешни Конфигурации во Spring

Надворешни конфигурации во Spring се техники што овозможуваат конфигурација на апликацијата преку внешни извори како што се property файлови (пр. `resources/application.properties`), environment променливи, конзолни аргументи, и сл. Ова овозможува динамичка и безбедна конфигурација без потреба за промена во изворниот код.

Надворешните конфигурации во Spring се силно средство за динамичка и безбедна конфигурација на апликациите. Апликацијата станува по-флексибилна и по-лесна за управување.

Извори на Надворешни Конфигурации во Spring

Во продолжение се изворите на вредностите на својствата. Овие извори се подредени според приоритет. Ова значи, дека Глобалните поставувања на Devtools имаат предност пред сите останати наведени потоа во набројувањето кое следува.

- Глобални поставувања на Devtools во вашиот домашен директориум (`./spring-boot-devtools.properties` кога devtools е активен).
- `@TestPropertySource` анотации во тестовите.
- properties атрибут во тестовите. Достапно во `@SpringBootTest` и тест анотациите за тестирање на одреден дел од вашата апликација.
- Аргументи од командната линија.

- Properties од `SPRING_APPLICATION_JSON` (вградени JSON во околинска променлива или системска својство).
- `ServletConfig init` параметри.
- `ServletContext init` параметри.
- Java системски својства (`System.getProperties()`).
- OS environment променливи.
- Апликациски својства специфични за профилот надвор од пакуваниот JAR (`profileapplication-p.properties` и YAML варијанти).
- Апликациски својства специфични за профилот пакувани во JAR архивата (`profileapplication-p.properties` и YAML варијанти).
- Апликациски својства надвор од пакуваниот JAR (`application.properties` и YAML варијанти).
- Апликациски својства пакувани во JAR архивата (`application.properties` и YAML варијанти).
- `@PropertySource` анотации во `@Configuration` класите.
- Предефинирани свойства (поставени со поставување `SpringApplication.setDefaultProperties()`).

3.5.4 Конфигурациски профили во Spring

Spring овозможува механизам за групирање на конфигурациски вредности и бинови во таканаречени профили. Ова овозможува конфигурација на апликацијата во зависност од околната во која се извршува. На пример, може да имаме посебна конфигурација за развој, тест, продукција, итн. Профилите се активираат со поставување на `spring.profiles.active` својството во `application.properties` фајлот или преку некој од другите извори на конфигурации наведени во секцијата 3.5.3. `@Profile` анотацијата во Spring овозможува конфигурација на биновите во зависност од активниот профил на апликацијата.

Пример за користење на `@Profile` анотацијата за конфигурација на бин во различни профили:

```

1 @Profile("test")
2 @Component
3 public class TestDataSourceConfig { // ...
4 }

5
6 @Profile("!test")
7 @Component
8 public class DataSourceConfig { // ...
9 }
```

Активирањето на профилот може да се изврши во `application.properties` фајлот:

```
1 spring.profiles.active=test,jpa
```

Ова овозможува креирање на различни конфигурации за различни околности (профили) без потреба за промена во изворниот код. Кога се работи за конфигурација на апликација, ова е многу практично и ефикасно решение, затоа што конфигурациските вредности може да ги дефинираме во неколку фајлови (пр. `application.properties`, `application-test.properties`, `application-prod.properties`, итн.) и да ги активираме соодветните профили во зависност од околнината во која се извршува апликацијата. Како што беше споменато во претходната секција, важно е да се напомене дека во `application.properties` ги имаме предефинираните вредности, а во зависност од профилот може да ги препокриеме само оние вредности кои се разликуваат од предефинираните.

Уште едно сценарио во кое може да ги користиме профилите е кога имаме потреба од различни имплементации на некој интерфејс во зависност од барањата кои ги имаме од различни клиенти. На пример, ако имаме потреба од различни имплементации на интерфејсот `PaymentCalculator` во зависност од клиентите за кои треба да се пресмета плата, тогаш може да се дефинираат повеќе имплементациски компоненти анотирани со различни профили и да се активираат соодветните профили во зависност во зависност од клиентот кај кој е поставена апликацијата.

Сумарно, профилите овозможуваат голема флексибилност и елегантно управување со конфигурациите и биновите кои треба да се читаат во контејнерот за инверзија на контролата во различни окolini.

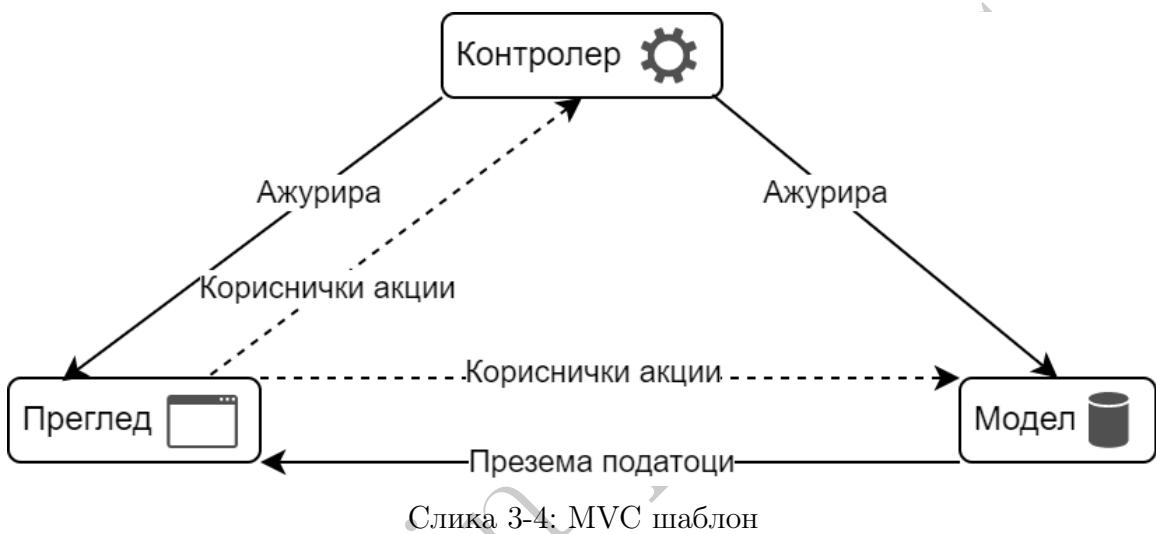
3.6 Шаблони за флексибилен и одржлив дизајн на веб апликации

Во ова поглавје ќе ги објасниме најчесто користените шаблони кои се неопходни за креирање на флексибилни, модуларни веб апликации, кои се лесни за одржување. Најпрвин ќе се запознаеме со SOLID принципите кои се клучни за овие карактеристики на сите апликации, а подоцна ќе се запознаеме со модел-предлед-контролер шаблонот (model-view-controller (MVC) pattern, англ.) кој ги користи и проширува овие принципи во контекстот на структурирање на кодот во про-

ектите на веб апликациите.

3.6.1 MVC шаблон

Модел - Преглед - Контролер шаблонот (Model-View-Controller (MVC) pattern, англ.) прикажан на слика 3-4 иницијално е креиран за десктоп апликации. MVC шаблонот го дели презентацијскиот слој во повеќе компоненти. Целта на овој шаблон е секоја од компонентите да има посебна одговорност и улога. На овој начин се постигнува поголема читливост на кодот, негово полесно одржување и тестирање.



Каде овој шаблон, улогите на компонентите се следните:

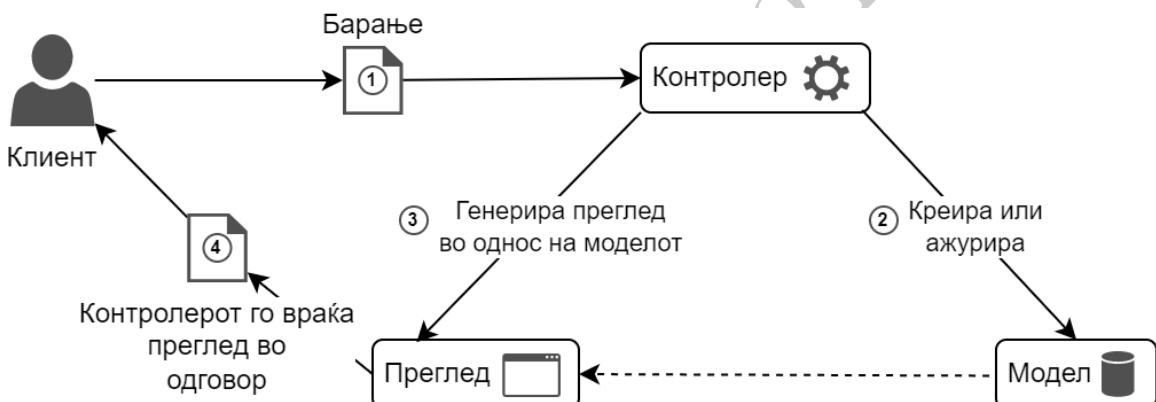
- Моделот ги содржи податоците кои се прикажуваат во прегледот.
- Прегледот ги содржи дефинициите на елементите кои се прикажуваат на крајниот корисник. Тука се дефинираат визуелните елементи, како што се табели, елементи за внес на податоци, копчиња, итн. При дефиницијата на прегледот се поставуваат зависности до моделот за динамичко прикажување на различни податоци.
- Контролерот е компонентата која е одговорна за справување со акциите на корисниците, како внес на податоци или клик на копче. Контролерот најчесто го менува моделот, но може да го промени и прегледот кој се користи (навигација помеѓу различни прегледи).

Во наједноставна форма, класичната имплементација на MVC шаблонот подразбира циклично извршување на акции и испртување на прегледот: животен циклус на кориснички интерфејс (UI life-cycle, англ.). При секоја корисничка

акција се повикува контролерот да ја изврши потребната бизнис логика и да го промени моделот. Потоа, во фазата на исцртување на прегледот, се користат новите вредности од моделот за промена на корисничкиот интерфејс.

3.6.2 MVC шаблон за веб апликации

Стандардниот MVC шаблон не е применлив во веб околината, поради природата на HTTP протоколот, за кој ќе стане збор во следната глава. Веб апликациите ги трансформираат податоците добиени во HTTP барањето во HTTP одговор. Притоа, цртањето на корисничкиот интерфејс го врши прелистувачот (browser, англ.) според податоците добиени во HTTP одговорот за соодветната страна. Секоја корисничка интеракција од гледна точка на веб апликацијата е репрезентирана со ново HTTP барање. Ова значи дека наместо да иницираме промени во прегледот кои автоматски ќе се исцртат, треба да ги побараме и превземеме промените од серверот.



Слика 3-5: MVC шаблон за веб апликации

На слика 3-5 е прикажана модификацијата на MVC шаблонот за веб апликациите. Главната разлика со стандардниот шаблон е во тоа што крајниот корисник комуницира директно со контролерот преку HTTP протоколот. Во MVC шаблонот за веб апликации, улогата на моделот останува непроменета, но улогите на контролерот и прегледот се делумно променети, поради различните презентацииски технологии:

- Контролерот е одговорен за:
 - извлекување на податоците добиени во HTTP барањето,
 - креирање, вчитување и зачувување на моделот,
 - извршување на бизнис логиката и промена на моделот,
 - селекција на прегледот,

- генерирање на содржината на HTTP одговорот со комбинирање на селектираниот поглед и модифицираниот модел.
- Прегледот најчесто е шаблон (template, англ.) кој се користи за генерирање на корисничкиот интерфејс.

3.6.3 Слоевита архитектура

Контролерот во MVC шаблонот за веб апликациите има голем број на одговорности. Имплементацијата на секоја од овие одговорности во една компонента нарушува поголем дел од SOLID практиките. Очигледно е дека со ваква имплементација не се следи практиката за една одговорност на компонентата. Дополнително, не може да се следи принципот за отвореност за проширување - затвореност за менување, бидејќи целата логика би била на исто место и секоја промена би барала промена директно во контролерот.

Со цел да се следат SOLID практиките, во организацијата на кодот се практикува воведување на слоеви. Концептот на слоеви е доста широко распространет во сите сфери од компјутерското инженерство, од компјутерските архитектури, преку компјутерските мрежи, па сè до софтверските системи. Неколку од главните придобивки од слоевитиот дизајн се следните:

- апстракција на однесувањето,
- поделба на одговорностите,
- лесна замена на компоненти од еден слој со нови понапредни и поефикасни компоненти,
- поедноставно тестирање преку симулирање на повиците од слојот кој се користи.

Најчестите слоеви кои се среќаваат при развојот на веб апликации се следните:

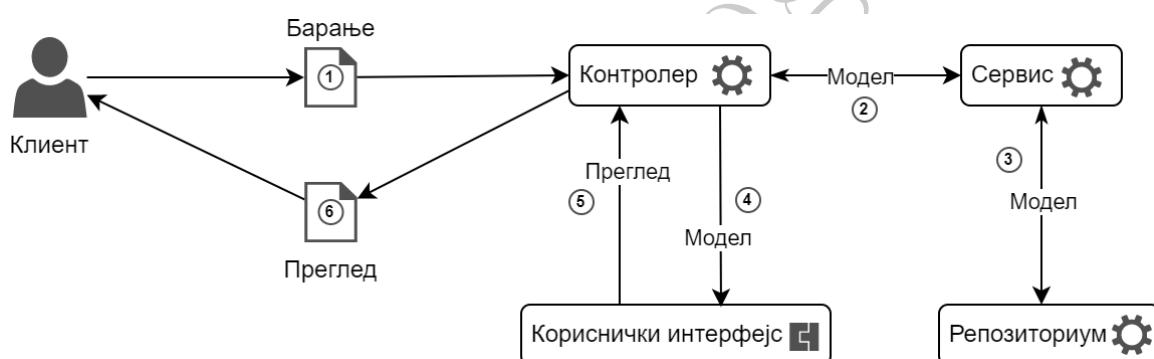
- **Модел:** репрезентација на бизнис проблемот што се решава. Дефинира што ќе се зачува како состојба во рамките на апликацијата. Оваа состојба најчесто се зачувува во некоја форма на податочно складиште, на пример во база на податоци. Многу важна напомена е дека не треба да има зависности во модел слојот. Овие класи треба да зависат само од стандардните Јава библиотеки (за датуми, стандардни типови како Integer, Long, Double, итн.) и од останатите класи од моделот. Причината за ова е што моделот треба да ја надживее апликацијата и технологијата. Технологите напредуваат брзо, но бизнисот е многу поконстантен. Затоа, овој слој треба да овозможи негово користење за поддршка на бизнис процесите дури и со

многу понапредни технологии кои можеби во моментот не се присутни. Во литературата овој слој може да се сртне под називите Domain Model или Model Layer.

- **Кориснички интерфејс (преглед):** ова е слојот што ги содржи шаблоните кои дефинираат како ќе се генерира корисничкиот интерфејс. Овој слој е зависен од библиотеките кои ќе се користат за процесирање на шаблоните кои ќе генерираат HTML или друг формат за приказ на информациите. Корисничкиот интерфејс е зависен од модел слојот и дефинира како податоците од овој слој ќе бидат прикажани на крајните корисници. Поради честите промени на преференците на корисниците, овој слој најчесто се менува. Оттука, целта при дизајнот е прегледот да не содржи бизнис логика и да може лесно да се замени. Бизнис логиката ги вклучува правилата за приказ, како условно прикажување на податоци во зависност од корисничките привилегии. За да може овој слој лесно да се замени, треба бројот на вакви услови да се сведе на минимум. Во пошироката литература, овој слој се нарекува User Interface Layer или едноставно се користи кратенката UI.
- **Презентативски слој (веб слој):** слојот кој што е задолжен за пакување (енкапсулација) на презентативската технологија и за интеграција помеѓу сервисниот и слојот за корисничкиот интерфејс. Ова е слојот кој ги извлекува параметрите од акциите на корисниците и ги делегира на сервисната логика, по што резултатите ги испраќа на корисничкиот интерфејс. Овој слој не треба да има бизнис логика во себе. Во контекст на веб апликациите, ова е единствениот слој кој има интеракција со HTTP протоколот и концептите од Java контејнерот. Ова е слојот кој е задолжен за извлекување на податоци од HTTP барањето (од параметрите, колачињата, заглавјата и од атрибутите на веб контејнерот) и за проследувањето на генерираните кориснички интерфејс во одговорот. Во литературата овој слој може да го сртнеме под називите Presentation Layer или Web Layer.
- **Сервисен слој:** ја дефинира бизнис логиката, односно го моделира однесувањето во рамките на бизнис процесите. И во овој слој не се препорачуваат зависности кон надворешни библиотеки. Овој слој треба единствено да зависи од класите од слојот за податочен пристап и од останатите класи од сервисниот слој. Единствена технолошка зависност која ќе ја дозволиме во сервисниот слој во рамките на оваа книга ќе биде Spring Core. Ова е техничкиот долг кој ќе го прифатиме за да ги добиеме бенефициите од инвертирање на контролата и вметнување на зависностите. Дополнително, препорачливо е сервисите да бидат без-состојбени, т.е. да не чуваат состојба. Тие треба да го користат слојот за податочен пристап за читање и запишување на потребните податоци. Во литературата овој слој се среќава

под називот Service Layer.

- **Слој за податочен пристап:** задолжен за апстракција на манипулацијата со податоците (зачувување, бришење, уредување и преземање). Погенерална дефиниција би била дека слојот за податочен пристап претставува апстракција на комуникацијата со други процеси. Најчесто другиот процес со кој комуницираме е база на податоци. Комуникацијата со веб сервиси исто така треба да биде апстрахирана преку слојот за податочен пристап. Најчесто, најголемиот дел од времето на извршување на веб апликацијата е токму кај кодот од овој слој. Токму затоа, апстракцијата на овој слој овозможува поедноставна промена на надворешниот сервис кој го користиме со цел подобрување на перформансите. Пример за ваква оптимизација на перформансите може да биде промена на базата на податоци од релациони бази во некој друг тип, при што ќе се запази имплементацијата на интерфејсите од овој слој. Во литературата овој слој се среќава под називите Persistence Layer, Data Access Layer или Repository Layer.



Слика 3-6: Слоевит MVC шаблон за веб апликации

3.6.4 Слоевит MVC шаблон за веб апликации

Слика 3-6 ја дава целосната слика за поврзување на MVC шаблонот за веб апликации со слоевите описаны во поглавјето 3.6.3. Како што може да се види од сликата, контролерот од MVC шаблонот ги содржи слоевите за презентација, сервис и податочен пристап. Меѓутоа, со ваквата организација се овозможува следење на SOLID принципите при развојот на самите веб апликации и се постигнува состојба во која секоја од компонентите има јасно дефинирана одговорност.

3.7 Преглед на основните концепти на Spring рамката

Во ова поглавје се запознавме со основните концепти на Spring рамката и нејзината дизајн филозофија, која е базирана на вметнување на зависностите преку инверзија на контролата, со што се овозможува брз и лесен развој на апликации кои лесно ќе се одржуваат во иднина. Притоа, ги разгледавме SOLID принципите кои ја поддржуваат оваа филозофија и ги објаснивме нивните предности.

Потоа, се запознавме со начините на конфигурирање на Spring биновите и со начините на внедрување на зависности помеѓу биновите. На крај, се запознавме со начините на конфигурирање на Spring биновите со помош на анотации.

На крај направивме преглед на шаблоните за флексибilen и одржлив дизајн на веб апликации, каде што ги објаснивме концептите на MVC шаблонот и слоевитата архитектура. Потоа, ги објаснивме како овие концепти може да се применат во контекстот на Spring веб апликации.

Знаењето стекнато во ова поглавје е неопходно за да се разбере функционирањето на Spring рамката, како и за тоа како да дизајнираме и развиеме апликации со чист, добро структуриран и разбиралив код кој е лесен за одржување. Концептите разработени во ова поглавје се нешто кое најчесто се покрива во интервјуа за работа за позиции во областа на развојот на софтвер со Јава програмскиот јазик и Spring рамката.

Глава 4

Spring MVC

Во ова поглавје, ќе ги обработиме деталите за Spring MVC со акцент на функционалностите од `org.springframework.web.servlet.DispatcherServlet`. Ќе започнеме со преглед на начинот на обработка на барањата и ќе ги идентификуваме компонентите кои играат клучна улога во справувањето со барањата. Откако ќе ги идентификуваме, ќе се запознаеме со начинот на креирање и конфигурирање на контролери со акцент на анотациите кои се користат за оваа намена. Исто така, ќе ги обработиме и начините за конфигурација на овој модул.

4.1 Запознавање со Spring MVC

Spring Web MVC е модул изграден врз JEE Servlet API и е вклучен во Spring рамката од самиот почеток. Како и многу други веб MVC рамки, Spring MVC модулот е дизајниран за да обработува HTTP барања со користење на централен сервлет кој ги делегира барањата до контролерите и нуди дополнителни функционалности што го олеснуваат развојот на веб апликации. За да можеме да го вклучиме овој модул во рамките на Spring Boot апликациите, потребно е во `pom.xml` да ја додадеме зависноста дадена во изворниот код [1](#).

Рамката Spring Web MVC е дизајнирана околу `DispatcherServlet`. Оваа ком-

Listing 1 Зависност за вклучување на Spring MVC модулот

```
<dependencies>
    <-- ... -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

понента е имплементирана како стандарден Java EE сервлет, регистрирана во Java EE контејнерот за справување со барања на одредена патека (најчесто ги пресретнува сите барања од контејнерот). Овој сервлет ги енкапсулира сите одговорности на контролер слојот од перспектива на веб MVC шаблонот описан во поглавјето [3.6.2](#).

Како што видовме во претходното поглавје, сервлетите ги процесираат барањата преку `doGet` и `doPost` методите, во кои треба рачно да ги земеме податоците испратени од барањето, да ги конвертираме во соодветните типови, па дури тогаш да ги повикаме потребните сервиси. `DispatcherServlet` го олеснува развојот на веб апликациите затоа што ги извлекува неопходните податоци од барањата и ги пренасочува до соодветните методи за справување со барањата (`RequestHandler`). Кога се користи Spring MVC модулот, развиваите се фокусираат на дефинирање на методот за справување со барањата кои најчесто се методи анотирани со `@RequestMapping` анотацијата¹. Методите за справување со барањата најчесто се дефинираат во класи кои се анотирани со `@Controller` или `@RestController`. Овие класи се нарекуваат контролери. Претходно споменатите анотации се само еден начин да се конфигурира поврзувањето на методот за справување со барањата со параметрите на HTTP барањата, како што се патеките, параметрите и заглавјата. Оваа конфигурација се нарекува мапирање на методот за справување со барањата (`HandlerMapping`).

Сервлетот `DispatcherServlet` од Spring MVC е дел од презентацијскиот слој (или веб слојот, поконкретно). Една од главните одговорности на презентацијскиот слој е прилагодување на параметрите добиени од презентацијската рамка (Java EE во нашиот случај) за процесирање во сервисната логика. Парсирањето на параметрите е неизбежно при секое барање од страна на клиентот. Во Spring MVC се оптимизира ова често сценарио за да се овозможи брз и лесно одржлив развој на веб апликации. Механизмот кој се користи е разрешување на аргументите на методите (`MethodArgumentResolver`). Процесот на разрешување на аргументите на методите овозможува да се вметнат аргументите на методот за справување со барањата врз основа на типот на податоци кој се очекува или врз основа на анотации на аргументите како што се `@RequestParam`, `@RequestHeader`, `@PathVariable` и други.

Дополнително, `DispatcherServlet` е одговорен и за пронаоѓање на прегледите (View) кои ќе се искористат за секој од методот за справување со барањата . Дополнително, овој сервлет се справува со локализацијата и поставувањето на теми за изглед и овозможува поддршка за полесно користење на прикачените датотеки.

¹Или со композитните анотации изведени од неа, како што се `@GetMapping` и `@PostMapping`.

Во одредени сценарија, имаме потреба да искористиме специфични процесирања пред или после спроведувањето со барањето. За оваа цел во Spring MVC може да регистрираме пресретнувачи (`HandlerInterceptor`). Еден пример за пресретнувач е `org.springframework.web.servlet.mvc.WebContentInterceptor`, кој може да се искористи за означување на барања кои може да бидат кеширани. Овој пресретнувач ги додава заглавјата "`Cache-Control`", "`Pragma`" и "`Expires`" во одговорот, со цел да даде инструкции за контрола на кеширањето на прелистувачот.

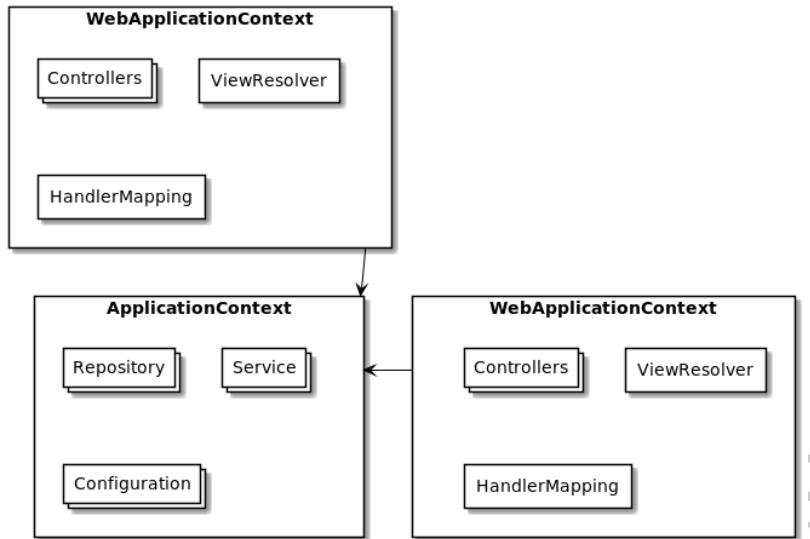
4.1.1 WebApplicationContext

`DispatcherServlet`-от користи контекст на веб апликацијата (`WebApplicationContext`) за сопствената конфигурација. Контекстот на веб апликацијата е проширување на обичниот апликациски контекст (`ApplicationContext`) од Spring рамката. Контекстот на веб апликацијата има референца до контекстот на сервлетот од JEE контејнерот и до сервлетот со кој е поврзан. Доколку имаме потреба, до контекстот на веб апликацијата можеме да пристапиме преку статичните методи на `RequestContextUtils` класата во рамките на кодот на веб апликацијата.

За секој `DispatcherServlet` кој го регистрираме во рамките на JEE контејнерот се креира посебен контекст на веб апликацијата кој има референца до основниот апликациски контекст. Основниот апликациски контекст вообичаено содржи инфраструктурни бинови, како што се конфигурации за складишта за податоци и деловни услуги кои треба да се споделат низ различни сервлети. Тие бинови ефективно се наследуваат и може да се заменат (односно, повторно да се декларираат) во специфичниот контекст на веб апликацијата за `DispatcherServlet`-от. Како што е прикажано на слика 4-1, контекстот на веб апликацијата обично содржи бинови специфични за дадениот `DispatcherServlet` како што се контролерите, мапирањата на методот за спроведување со барањата и конфигурациите за разрешувањите на прегледите (`ViewResolver`).

`DispatcherServlet`-от користи специјални бинови за да ги процесира барањата и да се генерираат соодветните одговори. Под „специјален бин“ мислиме на инстанци на објекти управувани од Spring рамката кои имаат специјални функции. Во продолжени се наведени некои од специјалните типови кои се користат од страна на `DispatcherServlet`-от:

- **HandlerMapping:** Ова е конфигурација која го содржи мапирањето на HTTP барањата до методот за спроведување со барањата. Во оваа конфигурација, за дадени барања може да има регистрирано и листа на пресретнувачи (`HandlerInterceptor`). Мапирањето се заснова на критериуми кои



Слика 4-1: Хиерархија на контексти кај Spring MVC

се разликуваат според имплементацијата на HandlerMapping. Двете најчести имплементации на HandlerMapping се RequestMappingHandlerMapping (која поддржува @RequestMapping анотирани методи) и SimpleUrlHandlerMapping (која одржува експлицитни регистрации на мапирања на шаблони од URL патеки до методот за справување со барањата). Во поглавјето 4.2 детално ќе ги разработиме критериумите за мапирањата на барањата со користење на анотирани методи.

- **HandlerAdapter:** Овие компоненти му помагаат на DispatcherServlet-от да го повика методот за справување со барањата кој е соодветен за тековното барање, без сервлетот да знае како точно треба да се повика компонентата. На пример, повикувањето на анотиран контролер бара соодветно процесирање на анотациите. Главната цел на HandlerAdapter е да го заштити DispatcherServlet-от од такви детали.
- **HandlerExceptionResolver:** HandlerExceptionResolver е стратегија за справување со исклучоците. Овозможува поврзување на исклучоците со методот за справување со барањата, со што може да имаме специфични прегледи за прикажување на грешните кон корисниците.
- **ViewResolver:** ViewResolver се користи за разрешување на логичките имиња на прегледи, кои се текстуални податоци вратени од методот за справување со барањата, во вистинскиот преглед (View) со кој ќе се генерира изгледот на страната во одговорот.
- **LocaleResolver, LocaleContextResolver:** Се користат за разрешување на јазикот што го користи клиентот и неговата временска зона за да може да се креираат интернационализирани прегледи. На овој начин може да се

користат различни текстуални податоци и слики во зависност од јазикот на клиентот т.е. да се користат повеќе-јазични ресурси.

- **MultipartResolver:** `MultipartResolver` е апстракција за справување на барања кои се составени од повеќе делови (`multipart`). Се користи библиотека за парсирање и процесирање на различните делови и нивно спојување во датотеките кои се прикачени преку форма на прелистувач (`file upload`).

4.2 Spring MVC анотирани контролери

Контролерите играат клучна улога во веб апликацијата: тие го извршуваат HTTP барањето, го подготвуваат моделот и избираат приказ за прикажување. Контролерот ги поврзува бизнис логиката и интерфејсот на апликацијата. Контролерот е компонента која е задолжена за одговор на акцијата што ја презема корисникот. Корисникот може да испрати податоци од форма, да кликне на линк или едноставно да пристапи до URL патеката на страницата. Контролерот ги избира или ажурира податоците потребни за прегледот. Исто така, го избира името на прегледот или може да го врати самиот преглед (`View`).

Со Spring MVC имаме две опции за дефинирање контролери: со имплементација на интерфејс или со поставање на соодветна анотација на класата. Интерфејсот кој треба да се имплементира е `org.springframework.web.servlet.mvc.Controller`, а анотацијата која се дава на класата за контролерот е `org.springframework.stereotype.Controller`. Главниот фокус на оваа книга е пристапот заснован на анотации (познат како *Spring @MVC*) за дефинирање на контролери.

Иако двата пристапа работат за имплементирање на контролер, постојат две големи разлики меѓу нив. Првата разлика е во флексибилноста, а втората е во мапирањето на URL патеките со контролерите. Контролерите базирани на анотации овозможуваат многу флексибилни потписи на методот², додека пристапот базиран на интерфејс има предефиниран метод што мораме да го имплементираме, дефиниран од интерфејсот. За пристапот базиран на имплементација на интерфејс, мора да направиме експлицитно надворешно мапирање на URL патеките за контролери преку соодветна конфигурација. Генерално, овој пристап е комбиниран со `org.springframework.web.servlet.handler.SimpleUrlHandlerMapping`, така што сите URL патеки се на една локација (каде што се конфигурирани). Тоа што сите URL патеки се на една локација е една од предностите на пристапот базиран на интерфејс. Кај пристапот заснован на анотации, мапирањата се расфрлани низ различни датотеки од кодот, што го отежнува пронаоѓањето на

²Потпис на методот претставува неговата видливост (`private`, `protected`, `public`), неговото име, аргументите кои ги прима и резултатот кој го враќа.

мапирањата на URL патеките со методот за справување со барањата ³. Предноста на контролерите базирани на анотации е тоа што, кога ќе се отвори еден контролер, можете да се види на кои URL патеки е мапиран секој од методите.

```

1 <dependencies>
2   <!-- ... -->
3   <dependency>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-thymeleaf</artifactId>
6   </dependency>
7 </dependencies>
```

Изворен код 4.1: Зависност за вклучување на Thymeleaf библиотеката

Во програмскиот модел базиран на анотации, класите анотирани со `@Controller` и `@RestController` користат дополнителни анотации за специфицирање на поврзувањето на патеките со методот за справување со барањата , за аргументите на барања, за справување со исклучоци и многу повеќе. Анотираниите контролери имаат флексибилни потписи за методите и не мора да ги прошируваат основните класи од Spring рамката, ниту да имплементираат специфични интерфејси.

```

1 @Controller
2 public class HelloController {
3
4   @GetMapping("/hello")
5   public String handle(Model model) {
6     model.addAttribute("message", "Hello World!");
7     return "index";
8   }
9 }
```

Изворен код 4.2: Едноставен контроллер

Изворниот код 4.2 покажува пример за контролер дефиниран со анотации. Во овој пример, методот има аргумент од тип `Model` и го враќа името на прегледот како `String`. Поконкретно, во овој метод за справување со барање се додава `"message"` атрибут во моделот со вредност `"Hello World!"`, кој треба да се прикаже во прегледот (`View`) именуван како `"index"`. Очигледно е дека во овој метод не е наведена релацијата со HTTP барањето и одговорот, меѓутоа Spring MVC ги

³Сепак, алатките како IntelliJ IDEA овозможуваат процесирање на целиот код и приказ на сите патеки со кои се справуваме во рамките на апликацијата.

определува аргументите на методите според нивниот тип или според анотациите наведени за соодветниот аргумент. Од друга страна, резултатот од овој метод се користи за пронаоѓање на прегледот (View). Spring MVC користи компонента наречена разрешувач на преглед (*ViewResolver*) за да го пронајде погледот за даденото име. Во Spring Boot има веќе конфигуриран *ViewResolver* кој ги бара HTML шаблоните во датотеката `classpath:/templates` и ја користи *Thymeleaf* синтаксата за процесирање на HTML шаблоните кои се користат како прегледи . Оваа конфигурација се овозможува автоматски со додавање на зависноста од изворниот код [4.1](#). Во рамки на Spring Boot проектите, сè што е поставено во директориумот `resources` се копира на локација `classpath:/`. Оттука, кога ја развиваме апликацијата, HTML шаблоните треба да ги поставиме во `resources/templates`. Откако ќе го пронајде, прегледот се исцртува со помош на атрибутите додадени во моделот и резултатот автоматски се вметнува во излезниот стрим (`OutputStream`) на HTTP одговорот. Доколку на патеката `/resources/templates/index.html` постои датотека со содржина дадена во изворниот код [4.3](#), тогаш ќе се генерира HTML страница како одговор, во која делот од содржината `th:text="${message}"` ќе се замени со вредноста на атрибутот `message` од моделот, т.е. со `"Hello World!"`. За да пристапиме до атрибутите од моделот во рамки на шаблоните со *Thymeleaf* синтакса, треба истите да ги референцираме со `${modelAttributeName}`. Постојат многу други опции за дефинирање на аргументи и резултати од методите кои ќе бидат објаснети подоцна во ова поглавје.

```

1  <!DOCTYPE HTML>
2  <html xmlns:th="http://www.thymeleaf.org">
3      <head>
4          <title>Title</title>
5      </head>
6      <body>
7          <div>Web Application. Passed model attribute:
8              th:text="${message}"</div>
9      </body>
</html>

```

Изворен код 4.3: `classpath:/templates/index.html`

Анотацијата `@Controller` овозможува автоматско откривање на каласите на контролерите, која е идентична со општата поддршка на Spring за откривање на класите анотирани со `@Component` во класната патека и автоматско регистрирање на дефинициите на биновите за нив. За да се овозможи автоматско откривање на контролер биновите анотирани со `@Controller`, можеме да додадеме скенирање на компонентите (`@ComponentScan`) во Јава конфигурацијата на апликацијата. Во

Spring Boot, ова е стандардно овозможено без дополнителни конфигурации.

4.2.1 Мапирање со HTTP барањата преку анотацијата @RequestMapping

Пишувањето методи за справување со барањата може да биде предизвик. На пример, како да кажеме за кои HTTP барања треба да се повика методот за справување со барањата ? Во овој случај важни се повеќе фактори како што се URL патеката, HTTP методот што се користи (на пр., GET или POST), достапноста на параметрите или HTTP заглавијата, па дури и типот на содржина на барањето или типот на содржина што се произведува (на пр., XML, JSON или HTML).

Првиот чекор во пишувањето на методи за справување со барањата е да се постави анотацијата `org.springframework.web.bind.annotation.RequestMapping` на методот. Оваа анотација може да се постави и на класа (контролер) и на нивото на метод. Анотирањето на класите се користи за да се направи грубо мапирање на сите нејзини методи (на пример, URL префиксот за сите методи или типот на содржините кои се произведуваат од методите), а потоа се користи анотацијата на ниво на метод за дополнително да се дефинира кога да се изврши методот (на пр., барање GET или POST). Целта на овој дел од книгата е да научиме како да ги анотираме методите на контролерот за да се справуваат со одредни барања и да може да препознаеме со кои барања ќе се повика веќе анотиран метод.

Анотацијата `@RequestMapping` подржува својства за совпаѓање со барањето по:

- **URL патеката** преку `path` или `value` својствата на анотацијата.

Со `path` или `value` својствата на анотацијата `@RequestMapping` може да се специфицираат шаблони дефинирани со `PathPattern` за да специфицираат за кои URL патеки ќе се повикаат анотираните методи. Следните три начини на користење на анотацијата се идентични: `@RequestMapping("/products")`, `@RequestMapping(value = "/products")` и `@RequestMapping(path = "/products")`. Исто така, можно е и специфицирање на повеќе патеки преку иста анотација: `@RequestMapping(path = "", "/products")`

`PathPattern` ја проширува синтаксата на `AntPathMatcher` која се користи во Spring MVC верзиите постари од 5.3. `PathPattern` ги процесира URL патектите користејќи ги следниве правила:

- `?` одговара на еден знак
- `*` одговара на нула или повеќе знаци во сегментот на патеката. Сегмент на патека е делот помеѓу две коши црти (`"/segment/"`)

- `**` одговара на нула или повеќе сегменти на патека кои се наоѓаат на крајот на патеката. Не е дозволено користење на овој израз во средина на патеката (пр. `d"/**/i"` или `"/**.txt"`)
- `pringspring` се совпаѓа со сегмент од патеката и го доловува како променлива во патеката со име `"spring"`
- `pring:[a-z]+spring:[a-z]+` доколку елементот од патеката се совпаѓа со регуларниот израз `"[a-z]+"`, вредноста се зема како променлива во патеката (path variable) со име „`spring`“. До пронајдените променливи во патеката може да се пристапи во анонтираните методи преку аргументи анонтирани со `@PathVariable`
- `spring*spring` одговара на нула или повеќе сегменти од патеката кои се наоѓаат на крајот на патеката и се доловуваат на променливата во патеката со име `"spring"`

- **HTTP метод** преку `method` својството.

Преку ова свойство може да се дефинира еден или повеќе HTTP методи (GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE) за кои ќе се повика методот од контролерот. Доколку не се наведе ова свойство, анонтирианиот метод од контролерот ќе се повика за сите HTTP методи за кои ќе се совпаднат останатите специфични свойства. Пр. `@RequestMapping(method = RequestMethod.PUT, RequestMethod.PATCH)` ќе се повика за барањата кои се со HTTP методи PUT или PATCH.

Исто така, постојат варијанти на приспособени анотации според HTTP методот на `@RequestMapping`:

- `@GetMapping`
- `@PostMapping`
- `@PutMapping`
- `@DeleteMapping`
- `@PatchMapping`

Приспособените анотации се обезбедени затоа што, најчесто, повеќето методи на контролерите треба да се мапираат на еден специфичен HTTP метод. Од друга страна, со користењето на `@RequestMapping` може да се специфицира совпаѓање со повеќе HTTP методи, доколку се наведени повеќе вредности, или со сите HTTP методи, доколку не е наведена вредност за `method` аргументот на анотацијата. Дополнително, `@RequestMapping` анотацијата е потребна за да се изразат споделените мапирања на ниво на класа.

- **Параметри на барање** преку `param` својството.

Ова свойство може да содржи низа од изрази во стилот `@RequestMapping(param="myParam=myValue")`. Барањето ќе биде мапира-

но само ако се открие дека го содржи специфицираниот параметар со конкретната наведена вредност. Изразите може да се негираат со користење на операторот "!=" како во `@RequestMapping(param="myParam!=myValue")`. Поддржани се и изразите на стилот `@RequestMapping(param="myParam")`, при што имињата на наведените параметри мора да бидат присутни во барањето (дозволено е да имаат каква било вредност). На крај, изразите во стилот `@RequestMapping(param="!myParam")` покажуваат дека наведениот параметар не треба да биде присутен во барањето.

- **Заглавија на барање преку header свойството.**

Ова свойство може да содржи низа од изрази во стилот `@RequestMapping(header = "myHeader=myValue")`. Исто како правилата за `param` свойството, и за ова свойство важат истите правила за нееднаквост на вредност, присуствот на заглавје или отсуство на заглавје. Дополнително, кај `header` свойството се поддржува цокер знакот (*), но само за тип на содржина т.е. за заглавија како Accept и Content-Type. На пример: `@RequestMapping("/something", headers = "content-type=text/*")`.

- **Типови на содржини испратени во барањето** преку `consumes` својството.

Го стеснува примарното мапирање по типови на содржини испратени во барањето. Се состои од еден или повеќе типови на содржини од кои еден мора да одговара на типот наведен во Content-Type заглавјето на барањето. Изразите може да се негираат со користење на операторот "!=". На пример, "`!text/plain`" ќе се мапира за сите барања со тип на содржина различен од `"text/plain"`. Исто така, поддржува цокер знаци за типот на содржината (*). Пример: `@RequestMapping("/something", consumes = "text/*")`.

- **Типови на содржини кои се прифатени од клиентот** преку `produces` својството.

Го стеснува примарното мапирање по типови содржина што може да ги произведе мапираниот контролер или метод. Се состои од еден или повеќе типови содржини од кои еден мора да се избере преку преговарањето за содржината (Content negotiation) со користење на „прифатливите“ типови содржини на барањето. Обично тие се извлекуваат од заглавјето „Accept“ на барањето. Изразите може да се негираат со користење на операторот "!=". На пример, "`!text/plain`" ќе се мапира за сите барања со тип на содржина различен од `"text/plain"`. Исто така, поддржува цокер знаци за типот на содржината (*). Пример: `@RequestMapping("/something", produces = "text/*")`.

¹ `@Controller`

² `@RequestMapping("/products")`

```

3 class ProductController {
4
5     @GetMapping("/{id}")
6     public Product showProduct(@PathVariable Long id) {
7         // ...
8     }
9
10    @PostMapping
11    public String addProduct(@RequestParam String productName,
12                             @RequestParam(required="false") Long quantity) {
13        // ...
14    }

```

Изворен код 4.4: Користење на @RequestMapping

Ако `@RequestMapping` се користи на ниво на класа, се дефинираат споделени мапирања кои ќе се применат на сите методи од класата. Доколку се користи на ниво на метод, се дефинира специфично мапирање за повикување на тој метод. Така, во изворниот код 4.4, `@RequestMapping` анотацијата на класата означува дека сите нејзини анотирани методи ќе се повикаат само доколку HTTP барањето содржи `"/products"` како префикс на патеката, а останатиот дел се совпаѓа со шаблонот конфигуриран за методот. Класата `ProductController` содржи два методи кои се мапираат на барањата со патеки `d"/products/i"` и `"/products"`. Притоа, првиот метод `showProduct` се повикува само кога HTTP методот е GET и само ако на местото на променливата `di` е поставен податок кој може да се претвори во `Long`. Вакви примери се барањата `"/products/1"`, `"/products/-123"` и `"/products/7865"`, додека пак барањата од формат `"/products/p1"` ќе резултираат со HTTP статус код 400 (Bad Request) бидејќи `"p1"` не може да се конвертира во тип `Long` и да се постави како вредност на аргументот на методот. Ако пак патеката содржи дополнителни сегменти, како на пример `"/products/5/details"` ќе се генерира резултат со HTTP статус код 404 (Not Found). Дополнително, барањето GET `"/products"` ќе врати HTTP статусен код 405 (Method Not Allowed), бидејќи патеката `"/products"` е единствено мапирана на барања со HTTP метод POST кои ќе го повикаат `addProduct` методот од контролерот. Од друга страна, методот `addProduct` од контролерот ќе се повика само со HTTP POST метод на патеката `"/products"`, но под услов да биде проследен и задолжителниот параметар со име `"productName"` во телото на барањето или како URL параметар на барањето. Во овој случај, не е задолжително вклучувањето на параметарот `"quantity"` како дел од барањето.

Во следниот пример, прикажан во изворниот код 4.5, во првиот метод се

дефинирани две променливи во патеката што вредности се користат како аргументи на методот со анотацијата `@PathVariable`. Освен ако не се специфицира поинаку со користење на `@PathVariable(name="patternVariableName")`, името на аргументот од методот се поврзува со името на променливата искористена во шаблонот за патеката во анотацијата. Во вториот метод од овој изворен код, дополнително е прикажан пример каде во шаблонот се користи регуларен израз за специфицирање на форматот на податоците кои се соодветни за променливата во патеката. Во случаите кога не можат соодветните делови од патеката да се совпаднат со регуларниот израз, се враќа одговор со HTTP статус код 404 (Not Found).

```

1 @GetMapping("/owners/{ownerId}/pets/{petId}")
2 public String findPet(@PathVariable Long ownerId, @PathVariable Long petId)
3 {
4     // ...
5 }
6 @GetMapping("/{name:[a-z-]+}-{version:\\d\\,\\d\\.\\d}{ext:\\.[a-z]+}")
7 public void handle(@PathVariable String name, @PathVariable String version,
8     @PathVariable String ext) {
9     // ...
}

```

Изворен код 4.5: Мапирање на методи во контролерот

4.2.2 Аnotирани методи за справување со барања (Handler Methods)

Методите анотирани со `@RequestMapping` имаат флексибilen потпис и можат да избираат од опсегот на поддржани аргументи и вредности. Во продолжение ќе се запознаеме со поддржаните аргументи и со начинот на процесирање на вредностите кои се враќаат од анотираните методи на контролерот.

Аргументи на методите

Во продолжение ќе ги обработиме аргументите кои може да се користат во рамки на методите за справување со барања во анотираните контролери. Дел од овие аргументи ќе бидат наведени како тип, а останатите како анотација. Причината за тоа е што во Spring MVC аргументите се разрешуваат преку нивниот тип ако тоа е еднозначно. Во спротивно, треба да се наведе анотација со која ќе преци-

зира од каде точно да биде земена вредноста на тој аргумент. Кај аргументите наведени според тип, имињата на променливите не играат улога во поставувањето на вредноста, па може да се користат произволни вредности. Така на пример, доколку искористиме аргумент од тип `HttpSession`, без разлика како ќе го имenuваме, се вклучува сесијата на корисникот од соодветното барање.

Секој од методите за справување со барања кои се дефинирани во рамките на контролерот се повикува од страна на `DispatcherServlet`-от. Тоа означува дека секој од овие методи има предефинирано множество на објекти кои се иницијализирани од страна на Java EE контејнерот. Такви објекти се барањето и одговорот кои се проследуваат на `service` методот од `DispatcherServlet`-от преку кои може да пристапиме до сесијата, сервлет контекстот и влезните и излезните стримови од барањето и одговорот. Во овој контекст, `DispatcherServlet`-от е во можност да ги пронајде вредностите за типовите кои претходно ги сноменавме и се еднозначни. Дополнително, барањето содржи параметри, заглавја и колачиња до кои може да се пристапи преку едноставни анотации искористени во методите за справување со барања од контролерот.

Во продолжение се дадени типовите, анотациите и описот на правилата според кои се вметнува вредноста за овие аргументи.

- **javax.servlet.ServletRequest, javax.servlet.ServletResponse**

За секое барање постојат различни и единствени `ServletRequest` и `ServletResponse` објекти кои ги добива `DispatcherServlet`-от од страна на Java EE контејнерот. Оттука, може да се искористат аргументи од овие типови во методите на контролерите и истите да бидат еднозначно пронајдени и проследени при повикот на методот од страна на `DispatcherServlet`-от. Доколку имаме потреба, исто така е можно да се искористи и некоја од изведените класи за барањата и одговорите, како што се `HttpServletRequest`, `HttpServletResponse`, `MultipartRequest` или `MultipartHttpServletRequest`.

- **javax.servlet.http.HttpSession**

Доколку се искористи `HttpSession` како аргумент на методот, се вметнува тековната сесија за барањето. Доколку не е креирана сесија пред ова барање за клиентот, со повик на методот кој содржи `HttpSession` како аргумент сесијата ќе биде креирана. Затоа, овој аргумент никогаш не е `null`. Важно е да се напомене дека пристапот до сесијата не е безбеден за нишки (not thread-safe), па ако има сценарија во кои повеќе барања истовремено пристапуваат до сесијата на ист клиент, потребно е да се постави `synchronizeOnSession=true` при конфигурацијата на инстанцата на `RequestMappingHandlerAdapter`. Начинот на конфигурација на овој параметар

тар е надвор од доменот на оваа книга.

- **HttpMethod**

Доколку методот од контролерот е мапиран за повеќе HTTP методи, тој во одредени ситуации може да ни биде потребно да се знае кој е методот на барањето. Едниот начин да се добие методот е да се постави `HttpServletRequest request` како аргумент и да се повика методот `request.getMethod()`. Меѓутоа, поедноставниот начин е едноставно да се искористи аргумент со тип `HttpMethod`, чија вредност ќе се постави да биде HTTP методот од тековното барање.

- **java.util.Locale**

Доколку имаме потреба да локализираме/преведеме ресурси во рамките на контролерот, јазикот на кој ќе бидат локализирани може да го вметнеме преку типот `Locale`. Начинот на определување на јазикот за тековното барање ќе биде описан во поглавјето [4.4](#).

- **java.util.TimeZone, java.time.ZoneId**

Слично како и јазикот, за клиентот се определува временската зона, препрезентирана преку типовите `TimeZone` и `ZoneId`. Бидејќи истите може еднозначно да се определат, Spring MVC овозможува користење на овие типови како аргументи, чии што вредности ќе се разрешат во моментот на повикување на методот.

- **java.io.InputStream, java.io.Reader**

Доколку имаме потреба да пристапиме до необработените стримови од податоци од барањето, тоа може да го сториме со вметување на аргументи со `InputStream` или `Reader` типовите. Овој начин е идентичен како да пристапуваме до `request.getInputStream()` методот од Servlet API. Необработен стрим од податоци од барањето може да ни биде потребен во сценарија кога податоците се испраќаат на нестандарден формат, кој би требало самите да го испроцејсраме.

- **java.io.OutputStream, java.io.Writer**

Доколку имаме потреба да пристапиме до излезните стримови од податоци од одговорот, тоа може да го сториме со вметнување на аргументи од типот `OutputStream` и `Writer`. На овој начин може да генерираме одговори во различни формати, како што се Excel форматите или PDF. Во овие случаи, препорачливо е методот на не враќа ништо како резултат, односно да биде деклариран како `void`.

- **@PathVariable**

Во поглавјето [4.2.1](#) објаснивме за анотацијата `@RequestMapping` и нејзините својства. Таму видовме дека во рамките на `@RequestMapping` може да специфицираме шаблони за URL патеки кои содржат променливи.

Дефинираните променливи во патеката може да ги пристапиме во рамките на методот преку аргументи анотирани со `@PathVariable`. Доколку не се наведе `value` или неговото синоним својство `name` во рамки на анотацијата, тогаш името на аргументот ќе се третира како име на променливата во патеката. Ако не постои променлива во патеката со името на аргументот, се добива одговор со HTTP статус 500 (Internal Server Error), кој е последица на исклучокот `MissingPathVariableException`. Во случаите кога сакаме името на аргументот да се разликува од името на променливата во патеката, тогаш треба името на променливата во патеката да го наведеме како вредност на `value` или `name` својствата на `@PathVariable` анотацијата: `@PathVariable("path-var-name")name`, `@PathVariable(value="path-var-name")name` и `@PathVariable(name="path-var-name")name` се идентични.

Дополнително, типот на аргументите анотирани со `@PathVariable` може да биде произволен, при што Spring MVC ќе се обиде да направи конверзија на типот од `String` во соодветниот тип. Начинот на конверзија ќе биде обработен подетално во поглавјето [4.2.2](#).

- **@RequestParam**

Оваа анотација се додава на аргументите кои треба да пристапат до параметрите од барањето, вклучително и повеќеделни датотеки (multipart files). Стандардно, параметрите на методот што ја користат оваа анотација се **задолжителни**, но можеме да поставиме дека параметарот на методот е опционален со поставување на `@RequestParam(required = false)` или со декларирање на аргументот со `java.util.Optional` типот. Конверзијата на типот автоматски се применува ако типот на аргументот не е `String`. Декларирањето на типот на аргументот како низа или листа овозможува вметнување на повеќето вредности за исто име на параметар. Кога анотацијата `@RequestParam` се декларира како `Map<String, String>` или `MultiValueMap<String, String>`, без име на параметар наведен во анотацијата, тогаш мапата се пополнува со вредностите на сите параметри од барањето, без разлика на искористеното име на аргументот. Да напоменеме дека употребата на `@RequestParam` е опционална за едноставни вредности на параметрите.

Во изворниот код [4.6](#), методот ќе се повика само доколку е присутен параметар со име `"product"` во барањето. Во случај кога овој параметар не е испратен, или кога истиот нема да може да се конвертира во тип `int`, ќе биде вратен одговор со HTTP статус 400 (Bad Request). Од друга страна пак, параметарот со име `"categoryId"` е опционален и аналогно, аргументот може да има `null` вредност.

```
1 @GetMapping("/demo")
2 public String handle(
3     @RequestParam("product") int productId,
4     @RequestParam(required=false) Long categoryId,
5 ) {
6
7 }
```

Изворен код 4.6: Користење на `@RequestParam`

- **@RequestHeader**

За пристап до заглавјата на барањата. Вредностите на заглавјето се конвертираат во декларираниот тип на аргумент на методот. Кога се користи анотацијата `@RequestHeader` на аргументи од типовите `Map<String, String>`, `MultiValueMap<String, String>` или `HttpHeaders`, мапите се пополнуваат со сите вредности за заглавјата присутни во барањето. Аргументите кои се анотирани со оваа анотација се задолжителни, освен ако не се постави `@RequestHeader(required=false)` или доколку типот на аргументот не се постави да биде `java.lang.Optional`.

Во изворниот код 4.7, методот ќе се повика само доколку се присутни заглавјата `"Accept-Encoding"` и `"Keep-Alive"`. За заглавјето `"Keep-Alive"` ќе се изврши конверзија на типот од `String` во `long`, доколку е возможно, а во спротивен случај ќе биде вратен HTTP статус код 400 (Bad Request).

```
1 @GetMapping("/demo")
2 public void handle(
3     @RequestHeader("Accept-Encoding") String encoding,
4     @RequestHeader("Keep-Alive") long keepAlive) {
5     // ...
6 }
```

Изворен код 4.7: Користење на `@RequestHeader`

- **@CookieValue**

For access to cookies. Cookies values are converted to the declared method argument type. The following example shows how to get the value of the cookie with name JSESSIONID, if present in the request. Otherwise, HTTP status code 400 (Bad Request) will be returned.

За пристап до колачинија. Вредностите на колачинијата се конвертираат во декларираниот тип на аргументот на методот. Изворниот код 4.8 покажува како да се добие вредноста на колачето со име `"JSESSIONID"`, доколку е

присутно во барањето. Во спротивно, HTTP статус 400 (Bad Request) ќе биде вратена. Аналогно како и кај претходните анотации, за колачето да не е задолжително се користи `@CookieValue(required=false)` или `Optional` како тип на аргументот.

```
1 @GetMapping("/demo")
2 public void handle(@CookieValue("JSESSIONID") String cookie) {
3     //...
4 }
```

Изворен код 4.8: Користење на `@CookieValue`

- `org.springframework.ui.Model`, `java.util.Map`,
`org.springframework.ui.ModelMap`

За пристап до моделот што се користи во HTML контролерите за него-во ажурирање и е овозможен на шаблоните за приказ за исцртување на корисничкиот интерфејс. Вредноста на аргументите од овие типови се иницијализира автоматски со нивните предефинирани конструктори.

- **@ModelAttribute**

Кога сакаме да пристапиме до постоечки атрибут во моделот (инстанциран ако не е присутен) со примена на врзување на податоци и валидација. Контролерот може да повеќе методи анотирани со `@ModelAttribute`. Сите такви методи се повикуваат пред методите анотирани со `@RequestMapping` во истиот контролер. Методот анотиран со `@ModelAttribute` исто така може да се сподели меѓу контролорите, доколку е дефиниран во класа анотирана со `@ControllerAdvice`. Методите анотирани со `@ModelAttribute` имаат флексибилни потписи на методот. Тие ги поддржуваат истите аргументи како методите на `@RequestMapping`, освен самиот `@ModelAttribute` или што било поврзано со телото на барањето.

Во изворниот код 4.9, првиот пример покажува метод анотиран со `@ModelAttribute`. Вториот пример додава само еден атрибут. Можеме исто така да користиме `@ModelAttribute` како анотација на методите `@RequestMapping`, во кој случај повратната вредност на методот анотиран со `@RequestMapping` се толкува како модел атрибут. Ова обично не е потребно, бидејќи е стандардно однесување во HTML контролерите, освен ако повратната вредност е `String` што инаку би се толкува како име на поглед. `@ModelAttribute` исто така може да го приспособи името на атрибутот на моделот, како што покажува третиот пример.

1 `@ModelAttribute`

```

2  public void populateModel(@RequestParam Long categoryId, Model model) {
3      model.addAttribute(categoryRepository.findCategory(categoryId));
4      // add more ...
5  }
6
7  @ModelAttribute
8  public Category addCategoryToModel(@RequestParam Long categoryId) {
9      return categoryRepository.findCategory(categoryId);
10 }
11
12 @GetMapping("/accounts/{id}")
13 @ModelAttribute("myCategory")
14 public Category handle() {
15     // ...
16     return category;
17 }

```

Изворен код 4.9: Користење на `@ModelAttribute`

- **`@SessionAttributes`**

`@SessionAttributes` се користи за складирање на атрибути на моделот во HTTP сесијата. Тоа е анотација на ниво на класа што ги декларира атрибутите на сесијата кои се користат од специфичен контролер. Преку оваа анотација обично ги наведуваме имињата на атрибутите на моделот што треба да бидат транспарентно складирани во сесијата за користење во следните барања за пристап.

```

1  @Controller
2  @SessionAttributes("category")
3  public class EditCategoryForm {
4      // ...
5  @PostMapping("/category/{id}")
6  public String handle(Long id, SessionStatus status) {
7      // ...
8      status.setComplete();
9  }
10 }

```

Изворен код 4.10: Користење на `@SessionAttributes`

Во изворниот код 4.10 е даден пример за користење на анотацијата `@SessionAttributes`. Во овој пример, преку анотацијата кажуваме дека атрибутот со име "category" од моделот, ќе се чува во сесијата. На првото барање,

кога на моделот му се додава атрибутот со име "category" тој автоматски се зачувува во сесијата. Останува таму додека друг метод на контролерот не го искористи `SessionStatus` како свој аргумент, преку кој ќе го отстрани атрибутот со соодветното име од сесијата. Притоа, `SessionStatus` се вметнува како аргумент според неговиот тип, и тој е овозможен само во класи кои се анотирани со `@SessionAttributes`.

- **@SessionAttribute**

Кога имаме потреба до пристап до веќе постоечки атрибути од сесијата и кои се управуваат глобално (односно, надвор од контролерот — на пример, со филтер), можеме да ја користиме анотацијата `@SessionAttribute` на аргументите на методи од контролерот, како што е случајот во изворниот код 4.11. Овие аргументи може да имаат и вредност `null`, доколку вреноста на соодветниот атрибут во сесијата не е поставена. Во ваквите случаи, треба да означиме дека аргументот не е задолжителен преку `required` својството. Кога ни е потребно додавање или отстранување на атрибути на сесијата, може да се искористат аргументи на методите од типовите `org.springframework.web.context.request.WebRequest` или `javax.servlet.http.HttpSession`, преку кои може да ги манипулираме сесиските атрибути преку Servlet API. За привремено складирање на атрибутите од моделот во сесијата како дел од работниот тек на контролерот, може да се користи `@SessionAttributes`, како што опишавме претходно.

```

1 @RequestMapping("/")
2 public String handle(@SessionAttribute User user) {
3 // ...
4 }
```

Изворен код 4.11: Користење на `@SessionAttribute`

- **@RequestAttribute**

За пристап до атрибутите на барањето може да се користи анотацијата `@RequestAttribute`. Преку оваа анотација може да ги пристапиме атрибутите кои се додадени преку филтри или преку некој HandlerInterceptor. Ако вредноста на овој атрибут е опционална, треба да се означи со `required=false`.

```

1 @GetMapping("/")
2 public String handle(@RequestAttribute Manufacturer manufacturer) {
3 // ...
4 }
```

Изворен код 4.12: Користење на `@RequestAttribute`

- **RedirectAttributes**

Стандардно, сите атрибути на моделот ќе се додадат како параметри на пренасочената локација. Додавањето на сите атрибути од моделот како параметри на локацијата за пренасочување може да биде посакуваниот резултат ако моделот е подготвен специјално за пренасочување. Меѓутоа, во анотирани контролери, моделот може да содржи дополнителни атрибути, додадени за целите на рендерирање (на пример, вредности на паѓачкото поле). За да се избегне можноста таквите атрибути да се појавуваат во URL-то, методите анотирани со `@RequestMapping` може да декларираат аргумент од типот `RedirectAttributes` и да го искористат за да ги специфицираат точните атрибути што треба да ги бидат достапни за `RedirectView`. Ако методот навистина пренасочува, се користи содржината на `RedirectAttributes`. Во спротивно, се користи содржината на моделот.

Променливите во патеката на тековното барање автоматски се достапни при проширување на URL-адресата за пренасочување и не треба експлицитно да ги додавате преку `Model` или `RedirectAttributes`. Следниот пример покажува како да се дефинира пренасочување:

```

1 @PostMapping("/files/{path}")
2 public String upload(...) {
3     // ...
4     return "redirect:files/{path}";
5 }
```

Изворен код 4.13: Имплементација на редирекција

- **Multipart**

Со конфигурирањето на `MultipartResolver`, содржината на POST барањата со тип на содржина `multipart/form-data` се процесира и може да се користи преку стандардни аргументи во методите од контролерот. Притоа, датотеките можеме да ги вметнеме како `@RequestParam` анотирани аргументи од типот `MultipartFile`. Декларирањето на типот на аргументот како `List<MultipartFile>` овозможува вметнување на повеќе датотеки за исто име на параметар.

Кога анатацијата `@RequestParam` се декларира за аргументи од типот `Map<String, MultipartFile>` или `MultiValueMap<String, MultipartFile>`, без име на параметар наведен во анатацијата, мапата се пополнува со сите повеќеделни датотеки за секое дадено име на параметар.

- **@RequestBody**

За пристап до телото на HTTP барањето. Содржината на телото се претвора во деклариралиот тип аргумент на методот со користење на имплементации на `HttpMessageConverter`. Може да ја користиме анотацијата `@RequestBody` за да го прочитаме телото на барањето и да го десериализирате во објект преку `HttpMessageConverter`. Следниот пример користи аргумент `@RequestBody`:

```

1 @PostMapping("/accounts")
2 public void handle(@RequestBody Account account) {
3     // ...
4 }
```

Изворен код 4.14: Користење на `@RequestBody`

- **HttpEntity**

За пристап до заглавијата и телото на барањата. Телото се претвора со `HttpMessageConverter`. `HttpEntity` е повеќе или помалку идентичен со користењето на `@RequestBody`, но се заснова на објект од контејнер што ги изложува заглавијата и телото на барањата. Во следниот изворен код се покажува пример:

```

1 @PostMapping("/accounts")
2 public void handle(HttpEntity<Account> entity) {
3     // ...
4 }
```

Изворен код 4.15: Користење на `HttpEntity`

- **@RequestPart**

За пристап до дел од барањето со тип на содржина `multipart/form-data`. Нуѓи конвертирање на телото на делот со `HttpMessageConverter`.

- **Errors, BindingResult**

For access to errors from validation and data binding for a command object (that is, a `@ModelAttribute` argument) or errors from the validation of a `@RequestBody` or `@RequestPart` arguments. You must declare an `Errors`, or `BindingResult` argument immediately after the validated method argument.

За пристап до грешки од валидација и поврзување податоци за командни објекти (односно, аргумент од моделот `@ModelAttribute`) или грешки од валидација на `@RequestBody` или `@RequestPart`. Мора да декларираме аргумент `Errors` или `BindingResult` веднаш по валидираниот аргумент во методот.

- **Any other argument**

Ако аргументот на методот не се совпаѓа со некоја од претходните вредности описани овде и е едноставен тип (определено со `BeanUtils.isSimpleProperty`), аргументот се третира како `@RequestParam`. Во спротивно, ако типот не е примитивен, аргументот се третира како `@ModelAttribute`.

Повеќето аргументи споменати овде може да се користат по произведен редослед. Сепак, постои единствен исклучок од тоа правило: аргументот `org.springframework.validation.BindingResult`. Овој аргумент мора да следи по аргументот за кој користиме поврзување на параметрите од барањето.

Конверзија на типови на аргументи

Важен дел од поврзувањето податоци е конверзија на типови. Кога ќе добиеме барање, единственото нешто што го имаме се податоци од тип `String`. Меѓутоа, во реалниот свет користиме многу различни типови на објекти, не само репрезентации на текст. Некои анотирани влезни аргументи на методите од контролерот се добиваат како `String` (како што се `@RequestParam`, `@RequestHeader`, `@PathVariable` и `@CookieValue`) и може да бараат конверзија на типот ако аргументот е деклариран како нешто различно од `String`. Затоа, сакаме да ги конвертираме тие текстуални податоци во нешто што можеме да го користиме, па овде доаѓа конверзија на типови. Со Spring, постојат три начини да се направи конверзија на типови:

- **Конвертори (Converters)** Конверторите се користат да претворат еден Јава тип во некој друг Јава тип. На пример од `Long` во `java.util.Date`, или од `Integer` во `Color`, или од `String` во `Date`. Конверторите може да се користат во веб слојот или во било кој друг слој на кој му треба претворање на податоци.
- **Форматери (Formatters)** Форматерите се користат за претворање на `String` во друг Јава тип и обратно. Оттука, еден од типовите мора да биде `String`. Со форматерите не можеме, на пример, да дефинираме претворање од `Long` во `Date`. Примери на веќе постоечки форматери се `DateFormat`, за парсирање на `String` во `java.util.Date` и за форматирање како текст на `Date`. Дополнително, форматерите може да се локализираат, што ги прави соодветни за веб околините, како што се и Spring MVC апликациите.
- **Уредувачи на својства (Property editors)** Уредувачите на својства се стариот начин за конвертирање на својства. Сепак, во поновите верзии се препорачува користење на конвертери и форматери.

Во изворниот код 4.16 прикажуваме приспособен конвертор и форматер за типот `Price`⁴. Имајте предвид дека `Converter` овозможува само еднонасочна конверзија помеѓу произволни типови (од `String` во `Price` во нашиот случај), додека `Formatter` овозможува двонасочна конверзија, но еден од типовите мора да биде `String`.

```

1  @AllArgsConstructor
2  @Data
3  class Price {
4      private Double price;
5      private String currency;
6  }
7
8  class StringToPriceConverter implements Converter<String, Price> {
9
10     @Override
11     public Price convert(String from) {
12         String[] data = from.split(" ");
13         return new Price(Double.parseDouble(data[0]), data[1]);
14     }
15 }
16
17 class PriceFormatter implements Formatter<Price> {
18     @Override
19     public String print(Price price, Locale locale) {
20         return price.getPrice() + " " + price.getCurrency();
21     }
22
23     @Override
24     public Price parse(String text, Locale locale) throws ParseException {
25         String[] data = text.split(" ");
26         return new Price(Double.parseDouble(data[0]), data[1]);
27     }
28 }
```

Изворен код 4.16: Конверзија на аргументи со `Converter` И `Formatter`

Ако сакаме да користиме `org.springframework.core.convert.converter.Converter` или `org.springframework.format.Formatter` во Spring MVC, тогаш ние треба да ги додадеме со одредена конфигурација. `WebMvcConfigurer` има метод за оваа наме-

⁴Ја користиме анотацијата `@Data` на Lombok за да означиме дека се вклучени getter и stetter методите, како и празниот конструктор.

на. Методот `addFormatters` може да се оптовари за да регистрира дополнителни конвертори и/или форматери, како што е прикажано во изворниот код 4.17. `WebMvcConfigurer` ги дефинира методите за прилагодување на Јава базираната конфигурација за Spring MVC овозможена преку `@EnableWebMvc`.

```
1 @Configuration
2 public class WebConfig implements WebMvcConfigurer {
3
4     @Override
5     public void addFormatters(FormatterRegistry registry) {
6         registry.addConverter(new StringToPriceConverter());
7         registry.addFormatter(new PriceFormatter());
8     }
9 }
```

Изворен код 4.17: Конфигурација на Formatter и Converter

Дополнително, `ConversionService` може да се вметне каде било во нашата апликација и може да обезбеди конверзија на типот користејќи го методот на конвертирање, како што е прикажано во изворниот код 4.18.

```
1 @Autowired
2 ConversionService conversionService;
3
4 @PatchMapping("{id}/update-price")
5 public void updatePrice(@PathVariable Long id, @RequestParam String price) {
6     Price priceVal = conversionService.convert(price, Price.class);
7     //...
8 }
```

Изворен код 4.18: Користење на регистрирани Converter

Подржани повратни вредности на методите од контролерите

Покрај сите различни типови на аргументи на методите за справување со барања, тие може да имаат и неколку различни повратни вредности. Во продолжение ќе ги наведеме стандардните повратни вредности и анотации на методите за справување со барања во анотираните контролери:

- **ModelAndView**

На овој начин експлицитно го дефинираме приказот, моделот и, опционално, статусот на одговорот.

```

1  @GetMapping("/add-product")
2  public ModelAndView addProduct() {
3      ModelAndView modelAndView = new ModelAndView("product-form");
4      modelAndView.addObject("categories", categoryService.findAll());
5      return modelAndView;
6  }

```

Изворен код 4.19: Користење на ModelAndView

- **View**

Пример за View за рендерирање заедно со имплицитниот модел утврден преку командните објекти и методите анотирани со `@ModelAttribute`. Методот на контролерот исто така може програмски да го збогати моделот со декларирање на аргумент Model.

- **java.util.Map, org.springframework.ui.Model**

Атрибутите што треба да се додадат на имплицитниот модел, со името на погледот имплицитно одредено преку `RequestToViewNameTranslator`. Во наредниот пример (изворен код 4.20), резултатот кој го враќаме ќе биде искористен како модел при процесирањето на погледот. За логичко име на погледот ќе се искористи името на методот, што значи дека ќе се бара HTML шаблонот `resources/templates/editProduct.html`.

```

1  @GetMapping("/{id}/edit")
2  public Model editProduct() {
3      Model model = new Model();
4      model.addObject("categories", categoryService.findAll());
5      return model;
6  }

```

Изворен код 4.20: Користење на Model

- **@ModelAttribute**

Атрибут што треба да се додаде на моделот, со името на погледот имплицитно одредено преку `RequestToViewNameTranslator`. Имајте предвид дека `@ModelAttribute` е опционален и се претпоставува како да е поставен, доколку не се совпадне типот кој се враќа со ниту една од опциите кои се описаны овде. Аналогоно како и кај претходниот пример (изворен код 4.20), името на методот ќе се искористи како логичко име за погледот кој ќе треба да биде пронајден и исцртан.

- **String**

Кога методот враќа String, всушност го враќаме логичкото име на погледот

што треба да се пронајде од некоја од имплементациите на `ViewResolver` и да се користи заедно со имплицитниот модел, дефиниран преку методите анотирани со `@ModelAttribute`. Методите од контролерот дополнително може програмски да го збогатат моделот со декларирање на аргумент од тип `Model` и додавање на атрибути во него.

- **void**

Се смета дека методот со `void` повратен тип (или `null` повратна вредност) целосно се справил со одговорот, ако во исто време има `ServletResponse` или `OutputStream` аргумент или како да е анотиран со `@ResponseStatus`.

Ако ништо од горенаведеното не е точно, `void` повратниот тип може исто така да означи „одговор без тело“ за REST контролери или предефинирано име на приказ за HTML контролери. Интерфејсот `RequestToViewNameTranslator` определува логичко име на приказот кога не е експлицитно дадено такво име (кога моделот враќа `void`, `Model` или `Map`).

- **@ResponseBody**

Може да ја користиме анотацијата `@ResponseBody` на метод за враќање на серијализирана верзија од објекти во телото на одговорот. Серијализацијата се врши преку `HttpMessageConverter`. Ваков пример е прикажан во изворниот код 4.21:

```
1 @GetMapping("/accounts/{id}")
2 @ResponseBody
3 public Account handle() {
4 // ...
5 }
```

Изворен код 4.21: Користење на `@ResponseBody`

`@ResponseBody` е поддржан и на ниво на класа и во овој случај се наследува од сите методи на контролерот. Ова е ефектот кој се постигнува со `@RestController`, која што е композитна анотација означена со `@Controller` и `@ResponseBody`.

- **HttpEntity, ResponseEntity**

Повратната вредност што го одредува целосниот одговор (вклучувајќи ги заглавјата и телото на HTTP одговорот). Вредноста која се враќа треба да се конвертира преку имплементации на `HttpMessageConverter` и да се запише во одговорот. `ResponseEntity` е како `@ResponseBody`, но со статус и заглавја, како што се гледа во изворниот код 4.22.

```
1 @GetMapping("/something")
```

```

2  public ResponseEntity<String> handle() {
3      String body = ... ;
4      String etag = ... ;
5      return ResponseEntity.ok().eTag(etag).body(body);
6  }

```

Изворен код 4.22: Користење на ResponseEntity

- **HttpHeaders**

За враќање одговор со заглавја и без тело.

- **Која било друга вратена вредност**

Секоја вратена вредност што не се совпаѓа со некоја од претходните вредности и не е едноставен тип (`BeanUtils.isSimpleProperty`) се третира како имплицитен модел. Вредностите кои се едноставни типови остануваат нерешени.

Конфигурација на ViewResolver

Spring го одлучува кој поглед ќе се прикаже преку разрешувачите на погледи, кои ви овозможуваат да прикажувате модели во прелистувачот без да ја врзувате имплементацијата со одредена технологија за приказ. ViewResolver ги мапира имињата на погледите на вистинските прикази. И рамката Spring доаѓа со неколку разрешувачи на погледи, на пр. `InternalResourceViewResolver`, `BeanNameViewResolver` и неколку други.

Во изворниот код 4.23 е прикажана конфигурација на `InternalResourceViewResolver`, кој вистинските прикази ќе ги бара во датотеката `"/WEB-INF/view/"`, со тоа што ќе го бара логичкото име на погледот, со додаден суфикс `".jsp"`. Понајдените прикази ќе ги процесира со помош на приказот `JstlView`, кој знае како да ја трансформира соодветната синтакса за приказот во соодветен HTML одговор на кој ќе се применат атрибутите од моделот.

```

1 @Bean
2 public ViewResolver internalResourceViewResolver() {
3     InternalResourceViewResolver bean = new InternalResourceViewResolver();
4     bean.setViewClass(JstlView.class);
5     bean.setPrefix("/WEB-INF/view/");
6     bean.setSuffix(".jsp");
7     return bean;
8 }

```

Изворен код 4.23: Конфигурација на ViewResolver

Кога работиме со Spring Boot, `WebMvcAutoConfiguration` автоматски ги конфигурира `InternalResourceViewResolver` и `BeanNameViewResolver` во нашиот контекст на апликацијата. Исто така, додавањето на соодветниот стартер проект за синтаксата на шаблони заменува голем дел од рачната конфигурација што треба да ја направиме поинаку. На пример, со додавање на зависноста `spring-boot-starter-thymeleaf` во нашата `pom.xml` конфигурација, се вклучува Thymeleaf синтаксата за шаблоните за приказ и не е потребна дополнителна конфигурација:

```

1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-thymeleaf</artifactId>
4   <version>${spring-boot-starter-thymeleaf.version}</version>
5 </dependency>
```

Изворен код 4.24: Вклучување на зависност за thymeleaf

Оваа зависност го конфигурира бинот од тип `ThymeleafViewResolver` со името `thymeleafViewResolver` во нашиот контекст на апликација. Можеме да го замениме авто-конфигурираниот `ThymeleafViewResolver` со обезбедување на истоимен бин кој рачно ќе го конфигурираме во рамките на нашата апликација. Решавачот за преглед за Thymeleaf работи така што го опкружува името на погледот со префикс и суфикс. Стандардните вредности на префиксот и суфиксот се `"classpath:/templates/"` и `".html"`, соодветно. Spring Boot, исто така, обезбедува опција за промена на стандардната вредност на префиксот и суфиксот со поставување на својствата `spring.thymeleaf.prefix` и `spring.thymeleaf.suffix` во `application.properties`, соодветно.

4.3 Spring REST

Досега го обработивме градењето на класична веб-апликација: испраќаме барање до серверот, серверот го обработува барањето и го враќаме резултатот на клиентот, кој го прикажува финалниот изглед на страницата. Меѓутоа, во текот на последната деценија, начинот на кој градиме веб-апликации значително се промени. Сега имаме JavaScript и JSON/XML, кои овозможуваат веб-апликации базирани на AJAX каде податоците се праќаат од серверот кога ќе бидат достапни (push) и пренесуваат голем дел од логиката за однесување на клиентот, вклучувајќи валидација, рендерирање на делови од еcranот итн. Ова поглавје ќе го опише REST⁵ (Representational State Transfer), архитектонски стил кој дава упатства како програмерите да размислуваат за веб-ресурси наместо за страници и

⁵https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

како да ракуваат со нив.

REST програмските интерфејси (API) користат унiformни идентификатори на ресурси (URI) за адресирање на ресурсите. На денешните веб-страници, дизајните на URI варираат од ремек-дела кои јасно го пренесуваат моделот на ресурси на интерфјсот, како што е примерот со <http://api.example.com/computers/hp/pro-book-G840>, па се до оние кои се многу потешки за луѓето да ги разберат, како на пр. : <http://api.example.com/a8dd0-5953-a1e1-9f1c-3400211c9a66>

Клиентите мора да ја следат парадигмата за поврзување на Интернет и да ги третираат URI патеките како само-објаснувачки идентификатори. Дизајнерите на REST програмскиот интерфејс треба да создадат URI кои ќе го пренесат моделот на ресурси на REST програмскиот интерфејс на нивните потенцијални развивачи на клиенти. Пред да се ги дефинираме препораките, неколку зборови за URI форматот, бидејќи препораките претставени во овој дел се однесуваат на форматот на URI. *RFC 3986* ја дефинира генеричката URI синтакса како што е прикажано подолу ⁶:

URI = scheme ":"//authority "/"path ["?"query] ["#"fragment]

Косата црта (/) не треба да биде вклучена на крајот од URI идентификаторите. Косата црта (/) треба да се користи како раздвојувач на сегментите од идентификаторот за да се покаже хиерархиска врска. Треба да се користат цртички (-) за да се подобри читливоста на URI идентификаторот. Долните цртички (_) не треба да се користат во URI. Во патеките на URI идентификаторите треба да се претпочитаат мали букви. Екстензии на датотеки не треба да бидат вклучени во URI идентификаторите. Препораката е да се користи договорање за типот на содржините (content negotiation) преку Accepts заглавјето. Името на ресурсот се препорачува да биде множина, како на пр. <http://api.example.com/api/manufacturers>, за потоа сите акции кои се однесуваат на ресурсот да ги специфицираме преку HTTP методи и дополнувања на патеката. Пример: GET <http://api.example.com/api/manufacturers/1> може да се искористи за пристап до произведувачот со идентификатор 1. Поголем дел од овие препораки за форматот на URI патеките се имплементирани во изворниот код [4.25](#) каде ресурсот е изразен во множина, патеките не завршуваат на коса црта (/) и се користи хиерархиско пристапување до ресурсите. Во овој пример нема потреба од користење на цртички (-) во имињата на патеките, но доколку некој од ресурсите е составен од повеќе зборови, тогаш зборовите е препорачливо да се оделени со (-), како што во примерот со ShoppingCart е препорачливо да се постави на патеката /api/shopping-carts).

Како што беше наведено претходно, една од клучните цели на пристапот REST е користење на HTTP како протокол со цел да се избегне обликување

⁶Подетално за форматот на URL патектите е објаснето во [1.2](#).

на различен API интерфејс за секоја апликација што се развива. Оттука, треба систематски да користиме HTTP глаголи за да опишеме какви дејства се извршуваат на ресурсите и да ја олесниме работата на развиваците на клиенти во процесот на справувањето со повторливите CRUD⁷ операции.

```

1  @AllArgsConstructor
2  @RestController
3  @RequestMapping("/api/manufacturers", produces="application/json")
4  public class ManufacturerController {
5      private final ManufacturerService service;
6
7      @GetMapping
8      public Page<Manufacturer> find(
9          @RequestHeader(defaultValue = "1") Long page,
10         @RequestHeader(defaultValue = "25") Long size,
11         @RequestHeader(required = false) List<String> sortBy,
12         @RequestParam Map<String, String> filters) {
13     return this.service.find(page, size, sortBy, filters);
14 }
15
16     @GetMapping("/{id}")
17     public Manufacturer findById(@PathVariable Long id) {
18         return this.service.findById(id)
19             .orElseThrow(ManufacturerNotFoundException::new);
20     }
21
22     @ResponseStatus(HttpStatus.NO_CONTENT)
23     @DeleteMapping("/{id}")
24     public void deleteById(@PathVariable Long id) {
25         this.service.deleteById(id);
26     }
27
28     @PostMapping
29     public ResponseEntity<Manufacturer> save(
30         @RequestParam String name,
31         @RequestParam String address,
32         UriComponentsBuilder builder) {
33         return this.service.save(name, address)
34             .map(manufacturer -> ResponseEntity

```

⁷CRUD е кратенка од Create (Креирај), Retrieve (Превземи), Update (Промени), Delete (Избриши) операциите.

```

34     .created(this.toUri(manufacturer.getId(), builder)
35     .body(manufacturer))
36     .orElseGet(() -> ResponseEntity.badRequest().build());
37 }
38
39 private URI toUri(Long id, UriComponentsBuilder builder) {
40     return MvcUriComponentsBuilder.fromController(builder,
41         this.getClass()).path("/{id}")
42         .buildAndExpand(manufacturer.getId()).toUri()
43     }
44

```

Изворен код 4.25: Пример REST контролер со CRUD операции

Следните методи се препорачани за користење во REST архитектурниот шаблон:

- **GET** Методот GET се користи за преземање информации од дадениот сервер со користење на даден URI. Барањата што користат GET треба да враќаат само податоци и не треба да имаат никакво друго влијание врз податоците, односно не треба да ја менуваат состојбата на серверот (ниту база, ниту контексти, ниту сесија). Накратко, овие барања треба да се идемпотентни.
- **HEAD** Исто како GET, но ја пренесува само статус линијата и делот за заглавјата. Овој метод е погоден за да провериме дали има промени на некој ресурс, без да го преземаме.
- **POST** Барањето POST се користи за испраќање податоци до серверот, на пример, информации за продукти, поставување датотеки итн. со користење на HTML форми. Најчесто со овој метод се креираат нови ресурси на серверот.
- **PUT** Ги заменува сите тековни податоци на целниот ресурс со поставената содржина испратена во барањето. Најчесто се користи за уредување на ресурс според идентификатор, при што целосно се заменува содржината на ресурсот со нови (или делумно променети) податоци испратени од клиентот.
- **PATCH** Се користи за да се направат мали ажурирања на ресурсите, како дел од својствата. Во овој случај не се заменува целиот ресурс со нови својства. Пример за користење на овој метод може да биде промена на цена на продукт, каде идентификаторот на продуктот се испраќа како дел од URI патеката за ресурсот кој се менува, а новата цена се испраќа во телото на барањето.
- **DELETE** Ги отстранува сите тековни податоци за целниот ресурс дадени

од URI.

- **OPTIONS** Ги опишува опциите за комуникација за целниот ресурс. Најчесто овој метод се користи од клиентите за да проверат кои методи се дозволени за пристап до даден ресурс.

Разлики помеѓу PUT и PATCH: Барањето со метод PUT е наменето да ја замени целата содржина на ресурсот, додека барањето PATCH се користи за да се направат промени на дел од ресурсот.

Изворниот код [4.25](#) ја следи препораката за примена на HTTP методите и практично покажува како може истите да ги конфигурираме за секој од методите од контролерот.

При работа со RESTful API од голема важност е користењето на соодветните HTTP статусни кодови. Иако во секција [1.6](#) ги вовдовме основните статусни кодови при описот на протоколот HTTP, во продолжение ќе бидат прикажани најчесто користените статусни кодови и нивната намена од аспект на RESTful сервиси:

- **200 – OK:** Барањето е успешно испроцесирано.
- **201 – CREATED:** Креиран е нов ресурс и неговиот URI идентификатор е поставен во Location заглавјето од одговорот. Најчесто се користи како статусен код на POST барања за креирање на ресурси. Изворниот код [4.25](#) прикажува како практично може да се постави овој статусен код и како да се изгради URI идентификаторот кој ќе се постави во Location заглавјето. За оваа цел се користи `ResponseEntity` како резултат бидејќи преку него се добива најголема контрола врз резултатот што се враќа кон клиентот.
- **204 – NO CONTENT:** Ресурсот е успешно избришан. Се препорачува да се користи при бришење на ресурси со DELETE методот и не се очекува да се врати содржина во телото на одговорот. Во изворниот код [4.25](#) е поставен овој HTTP статусен код со помош на анотацијата `@ResponseStatus` со која наједноставно може да се конфигурира статусот кој ќе се врати од даден метод. Истото може да се постигне и со `ResponseEntity` како резултат на методот, но `@ResponseStatus` анотацијата овозможува поголема прегледност на кодот и е полесна за користење.
- **304 – NOT MODIFIED:** Се користи кога е овозможено кеширање на податоците. Со овој статусен код се укажува дека бараниот ресурс не е променет.
- **400 – BAD REQUEST:** Барањето е неважечко или не може да се испроцесира. Точната грешка треба да се лоцира во телото на грешката. На пример, „JSON не е валиден“. Најчеста причина за овој статус код се погрешно испратени параметри (испуштени или од погрешен тип) или заглавја во

рамките на барањето, односно погрешно повикување од страна на клиентот.

- **401 – UNAUTHORIZED:** Потребна е автентикација (најава) за да се испроцесира барањето за ресурсот.
- **403 – FORBIDDEN:** Серверот го примил и го разбира барањето, но одбива да го опслужи бидејќи клиентот нема привилегии за пристап.
- **404 – NOT FOUND:** Не постои ресурс за дадениот URI идентификатор. Иако не се гледа во изворниот код [4.25](#), овој статусен код е една од можностите за методот `findById` бидејќи на класата на исклучокот е поставена анотацијата `@ResponseStatus(code = HttpStatus.NOT_FOUND)`. На овој начин е овозможено на едноставен начин да се конфигурираат статусните кодови во случај на исклучоци во рамките на апликацијата.
- **500 – INTERNAL SERVER ERROR:** Програмерите на API интерфјеси треба да ја избегнат оваа грешка. Тоа значи дека тие не се справиле со дадено сценарио, т.е. направиле грешка. Овие грешки треба да се логираат и да се коригираат штом бидат детектирани.

Полињата на HTTP заглавјето треба да се користат за да се обезбедат потребните информации со цел барањето или одговорот да бидат разбрани, или да го доопишат објектот испратен во телото на пораката. Најчесто, заглавјата ја содржат автентикацијата или овластвувањето на клиентот. Дополнително, заглавјето `Location` се користи за да означи која е вистинската URL адреса на ресурсите кога се користи пренасочување, или каков ресурс е создаден во случај на HTTP статус 201. Во изворниот код [4.25](#), заглавјата од барањето се искористени за пренесување на информации за страничењето и сортирањето. **Страницењето** на ресурсите е неопходно да се предвиди уште во раната фаза на дизајнирање на програмскиот интерфејс. Навистина е тешко да се предвиди прецизно прогресијата на количината на податоци што ќе се вратат, па затоа е препорачливо да се користи страницење на ресурсите со стандардни вредности кога тие не се обезбедени од клиентот што повикува. На пример, во методот `find` од изворниот код [4.25](#) се користат предефинирани вредности на соодветните заглава за соопсег на вредности [0-25]. Заглавјето за **сортирање** треба да ги содржи имињата на атрибутите по кои се врши сортирањето, одделени со запирка. Во примерот, ова заглавје е опционално.

Друг важен дел од дизајнот на REST API е користењето на параметрите за пребарување. Тие се широко користени во многу случаи, но повеќејатно е дека се користат за да се постигне некаков вид **страницење (paging)**, **фильтрирање** и **сортирање**. Сепак, во рамките на методот `find` од изворниот код [4.25](#) се искористени заглавја наместо параметри за страницењето и сортирањето, што е всушност одлука при дизајнот на програмскиот интерфејс која не влијае премногу.

гу на перформансите и на начинот на неговото користење. **Филтерирањето** се состои во ограничување на бројот на бараните ресурси со наведување на некои атрибути и нивните очекувани вредности. Препорачливо е да се подржи филтрирање на колекции од ресурси по неколку атрибути истовремено и да се дозволат неколку вредности за еден филтриран атрибут. Во методот `find` од изворниот код [4.25](#) се користат сите параметри кои се испратени во барањето како потенцијални филтри бидејќи `@RequestParam` анотацијата е поставена на типот `Map<String, String>`. Всушност, потребата сите параметри да се третираат како филтри наложува користење на заглавја за испраќање на податоците за страничење и сортирање. Во спротивно не би можело да се направи разлика помеѓу параметрите за фитрирање и оние за страничење и сортирање.

4.3.1 Конзумирање на податоци преку REST API

Не е невообичаено за Spring апликациите да обезбедуваат API и да поднесуваат барања до API на друга апликација. Всушност, ова станува распространето во светот на микросервисите. Затоа, вреди да поминеме малку време за да видиме како да ја користите Spring рамката за да конзумираме REST API.

Spring апликациите може да конзумираат REST API со `RestTemplate` - единствен, синхрон REST клиент обезбеден од основната Spring рамка. Од Spring верзија 5, воведен е нов HTTP клиент наречен `WebClient`. `WebClient` е модерен, алтернативен HTTP клиент на `RestTemplate`. Не само што обезбедува традиционално синхроно API, туку поддржува и ефикасен неблокирачки и асинхрон пристап. Така, ако развиваме нови апликации или миграраме стара, добра идеја е да користиме `WebClient`. Меѓутоа, ќе се задржиме на стандардниот `RestTemplate` за да може да ги разбереме и апликациите кои се напишани со негово користење.

Има многу работи на кои треба да се внимава во интеракцијата со REST ресурси од перспектива на клиентот, што е воглавно мачна и здодевна работа за прилагодување на очекуваните параметри и заглавја. Ако се користат HTTP библиотеки на ниско ниво, клиентот треба да создаде инстанца клиент и објект за барање, да го изврши барањето, да го интерпретира одговорот, да го мапира одговорот во објекти од доменот и да се справи со сите исклучоци кои може да се фрлат при повиците. И сето ова се повторува, без оглед на тоа какво HTTP барање е испратено.

За да се избегне таков нечист и повторлив код, Spring го обезбедува `RestTemplate`. `RestTemplate` ги ослободува програмерите од пишување на повторлив код кој се однесува на воспоставување на конекции и парсирање на податоците добиени од REST програмските интерфејсите. `RestTemplate` обезбедува **41 методи** за интеракција со REST ресурсите. Наместо да ги разгледаме сите

поединечно, ќе ги класифицираме според начинот на користење и ќе разгледаме само десетина уникатни операции, секоја преоптоварена за да се изедначи со комплетното множество од 41 метод. 12-те операции се следните:

- `delete(...)`: Извршува HTTP DELETE барање на ресурс на одредена URL адреса
- `exchange(...)`: Извршува одреден HTTP метод во однос на URL патека, враќајќи `ResponseEntity` што содржи објект конвертиран од телото на одговорот. Овој метод е погоден доколку сакаме да го испроцесираме и статусниот кодот вратен од REST програмскиот интерфејс кој го повикуваме, како и неговите заглавја.
- `execute(...)`: Извршува одреден HTTP метод во однос на URL, враќајќи објект конвертиран од телото на одговорот.
- `getForEntity(...)`: Испраќа HTTP GET барање, враќајќи `ResponseEntity` што содржи објект конвертиран од телото на одговорот
- `getForObject(...)`: Испраќа HTTP GET барање, враќајќи објект конвертиран од тело на одговорот
- `headForHeaders(...)`: Испраќа HTTP HEAD барање, враќајќи ги заглавијата на HTTP за наведената URL адреса на ресурси
- `optionsForAllow(...)`: Испраќа HTTP OPTIONS барање, враќајќи го заглавјето Allow за наведената URL-адреса
- `patchForObject(...)`: Испраќа HTTP PATCH барање, враќајќи го добиениот објект конвертиран од телото на одговорот
- `postForEntity(...)`: Испраќа податоци преку POST барање на URL-адреса, враќајќи `ResponseEntity` што содржи објект конвертиран од телото на одговорот
- `postForLocation(...)`: Испраќа податоци преку POST барање на URL-адреса, враќајќи го URL-то на ново-kreираниот ресурс
- `postForObject(...)`: Испраќа податоци преку POST барање на URL-адреса, враќајќи објект конвертиран од телото на одговорот
- `put(...)`: Испраќа податоци преку PUT барање на URL-адреса

Со исклучок на TRACE, `RestTemplate` има најмалку еден метод за секој од стандардните HTTP методи. Покрај тоа, `execute()` и `exchange()` обезбедуваат методи од пониско ниво, за општа намена при испраќање барања со кој било HTTP метод. Повеќето од методите описаны претходно се преоптоварени во три форми на методи:

- Една која прима `String` за спецификација на URL, каде URL параметрите се специфицираат како променлива листа со аргументи (`params...`):

```
dManufacturer manufacturer = restTemplate.getForObject(
    "http://localhost:8080/api/manufacturers/i", Manufacturer.class, 1);
• Друга која прима String за спецификација на URL, каде URL параметрите се специфицираат преку Map<String, String>:
dManufacturer manufacturer = restTemplate.getForObject(
    "http://localhost:8080/api/manufacturers/i", Manufacturer.class,
    Map.of("id", "1"));
• Трета која прима java.net.URI за спецификација на URL, без поддршка за параметризирани URL:
dManufacturer manufacturer = restTemplate.getForObject(
    UriComponentsBuilder.fromHttpUrl("http://localhost:8080/api/manufacturers/i").build()
    "1"), Manufacturer.class);
```

Постојат два начини за користење на RestTemplate:

1. да се креира инстанца во моментот кога ни е потребен RestTemplate rest = new RestTemplate()
2. да се декларира како бин и да се вметнеме таму каде што ни е потребен (изворен код 4.26)

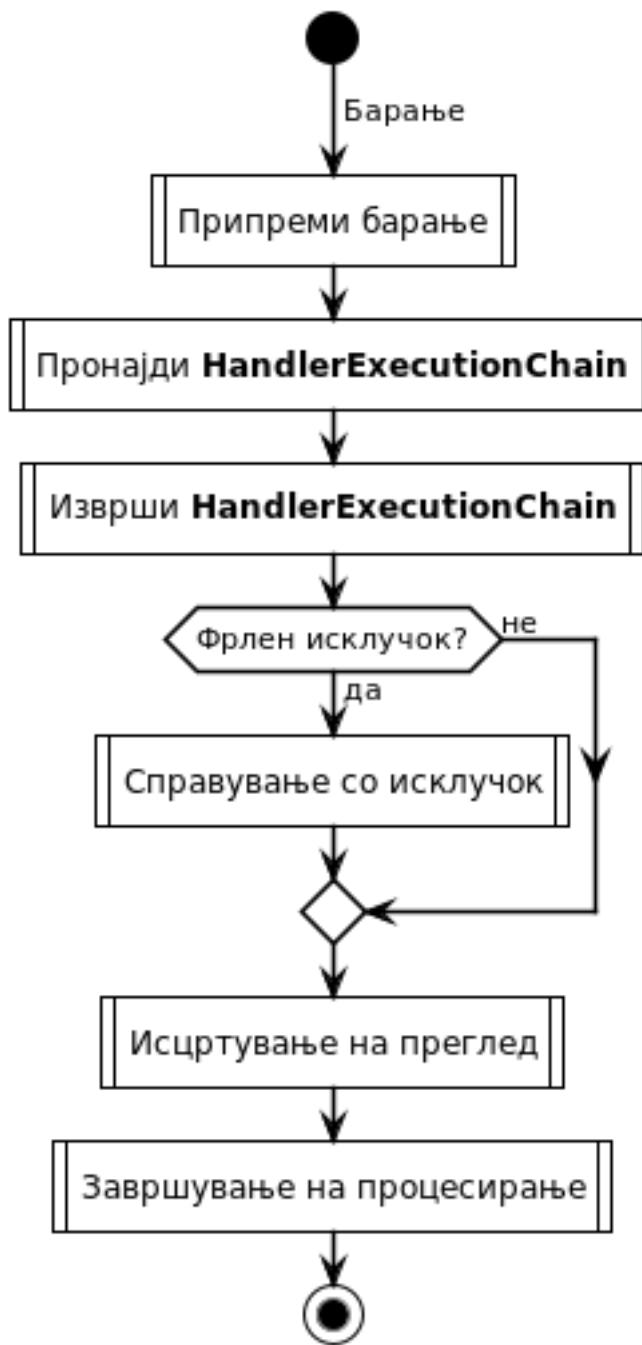
Препорачливо е да се употребува вториот начин на декларирање бидејќи може да се постават зеднички конфигурации кои ќе се споделат во сите повиди за дадено REST API. На овој начин, само еднаш ќе биде креирана и конфигурирана инстанцата на RestTemplate, со што се избегнува дуплицирање на код и потенцијални грешки во самиот тој процес.

```
1 @Bean
2 public RestTemplate restTemplate() {
3     return new RestTemplate();
4 }
```

Изворен код 4.26: Конфигурација на RestTemplate

4.4 Процесирање на барање кај Spring MVC

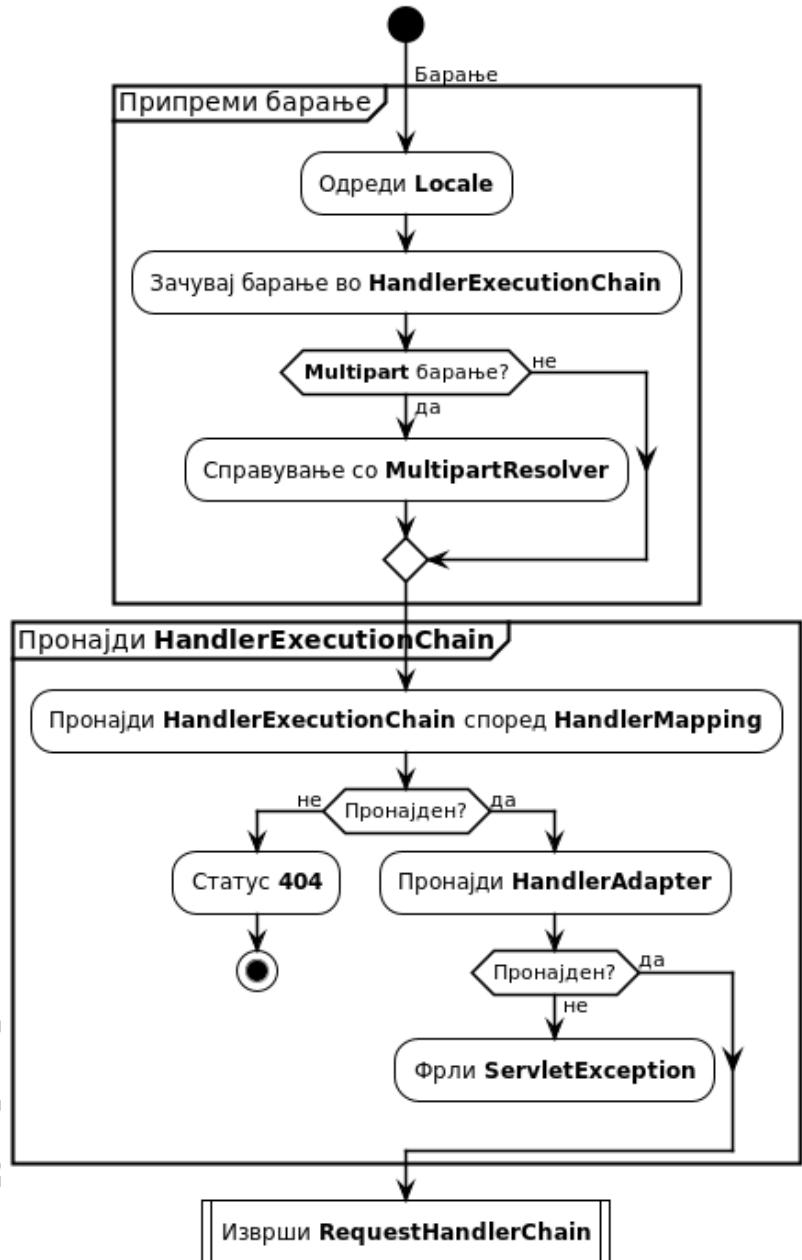
До сега се запознавме со начинот на дефинирање на контролери и како истите да ги мапираме за дадени барања од клиентите. Во продолжение ќе погледнеме како детално се процесира едно барање. На слика 4-2 е даден приказ на чекорите во процесирањето на барање од страна на Spring MVC, поконкретно, од страна на DispatcherServlet-от. Во овоа поглавје ќе ги обработиме сите овие чекори во детали.



Слика 4-2: Процесирање на барање кај Spring MVC

Повеќето делови од архитектурата на Spring поддржуваат интернационализација, исто како што тоа го прави и Spring web MVC. DispatcherServlet-от овозможува автоматски да се локализираат (да се преведат) пораките користејќи го јазикот на клиентот кој го испраќа барањето. Ова се прави со објектите `org.springframework.web.servlet.LocaleResolver`. Имплементациите на

LocaleResolver се дефинирани во пакетот `org.springframework.web.servlet.i18n` и може да се вклучат во контекст на Spring MVC апликација како стандардни бинови (со користење на методи анотирани со `@Bean` во конфигурациските класи).



Слика 4-3: Припрема на барање кај Spring MVC

Во првата фаза од процесирањето на барањето (прикажана на слика 4-3), DispatcherServlet-от извршува припреми кои го олеснуваат подоцнежното процесирање. Во првиот чекор, овој сервлет ја одредува локализацијата на барањето, претставена преку класата `java.util.Locale`. Овде, DispatcherServlet бара

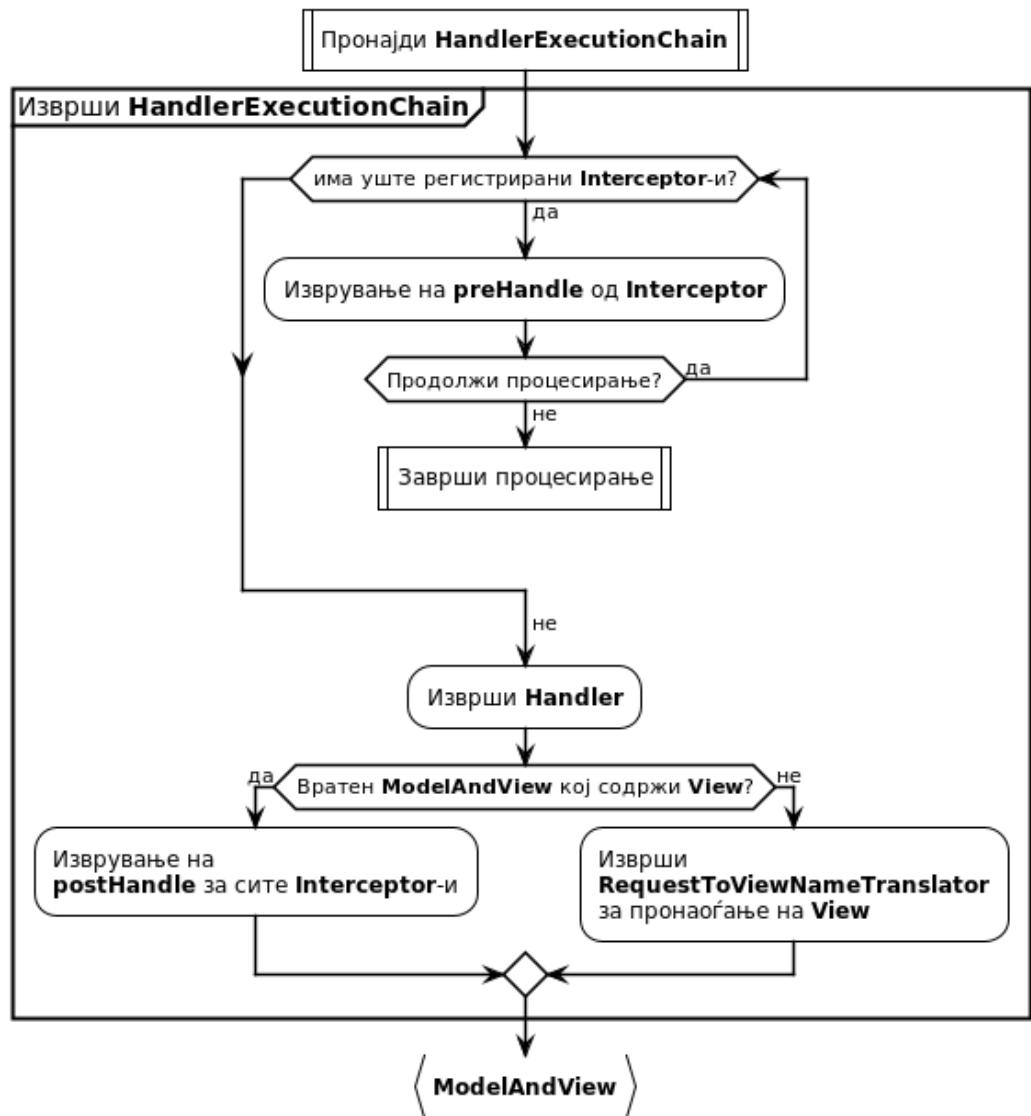
`LocaleResolver` во IoC контејнерот. Ако најде имплементација, ја искористи за да го определи јазикот кој ќе се користи за тековното барање.

Следно, `DispatcherServlet`-от креира `RequestContext` во кој се поставува тековното барање и претходно определениот јазик. За да може лесно да се пристапат информациите од тековното барање и тековниот јазик, без да се делегираат се-каде кај што е потребно, `DispatcherServlet`-от го поставува `RequestContext`-от во `org.springframework.web.context.request.RequestContextHolder`.

Следно, се проверува дали барањето е повеќеделно (multipart) HTTP барање (ова се користи кога се прикачуваат датотеки преку HTTP протоколот). Ако е така, барањето се проследува на `org.springframework.web.multipart.MultipartResolver` имплементацијата, која го трансформира во `MultipartHttpServletRequest`. По ова, барањето е подготвено да биде испратено до соодветната компонента за справување со барања (`RequestHandler`). Со користењето на `MultipartResolver` се олеснува начинот па пристапување и процесирање на прикачените датотеки.

Кога барањето е припремено, `DispatcherServlet` консултира една или повеќе имплементации на `org.springframework.web.servlet.HandlerMapping` за да одреди која компонента за справување со барањата (`RequestHandler`) е соодветна. Ако не се најде ваква компонента, се поставува HTTP статус код 404 и одговорот се испраќа назад до клиентот. Резултатот од `HandlerMapping` е `org.springframework.web.servlet.HandlerExecutionChain`. Кога е пронајден `HandlerExecutionChain`, серверот ќе се обиде да најде регистриран `org.springframework.web.servlet.HandlerAdapter` во `WebApplicationContext` за да го изврши `HandlerExecutionChain` кој е добиен. Ако не може да се најде соодветен `HandlerAdapter`, се фрла `javax.servlet.ServletException`. Како што веќе споменавме, `HandlerAdapter` се користи за да се апстрахираат деталите за повикувањето на соодветниот `RequestHandler`.

За да се спрavi со барањето, `DispatcherServlet` го користи `HandlerExecutionChain` за да одреди што треба да изврши. Оваа класа има референца за вистинската компонента за справување со барањето што треба да се повика, како и опционални референци кон имплементации од `org.springframework.web.servlet.HandlerInterceptor`, кои дефинираат што да се изврши пред (метод `preHandle`) и по (метод `postHandle`) извршувањето на компонентата за справување со барањето. Методот `preHandle` на пресретнувачите враќа дали треба да се продолжи понатаму со процесирање на барањето. Како што е покажано на Слика 4-4, се извршуваат сите пресретнувачи сè додека не се изминат сите или додека некој од нив не врати дека не треба да се продолжи со процесирањето на барањето. Ако се врати дека не треба да се продолжи, се извршува процедурата за завршување на процесирањето, која ќе биде описана

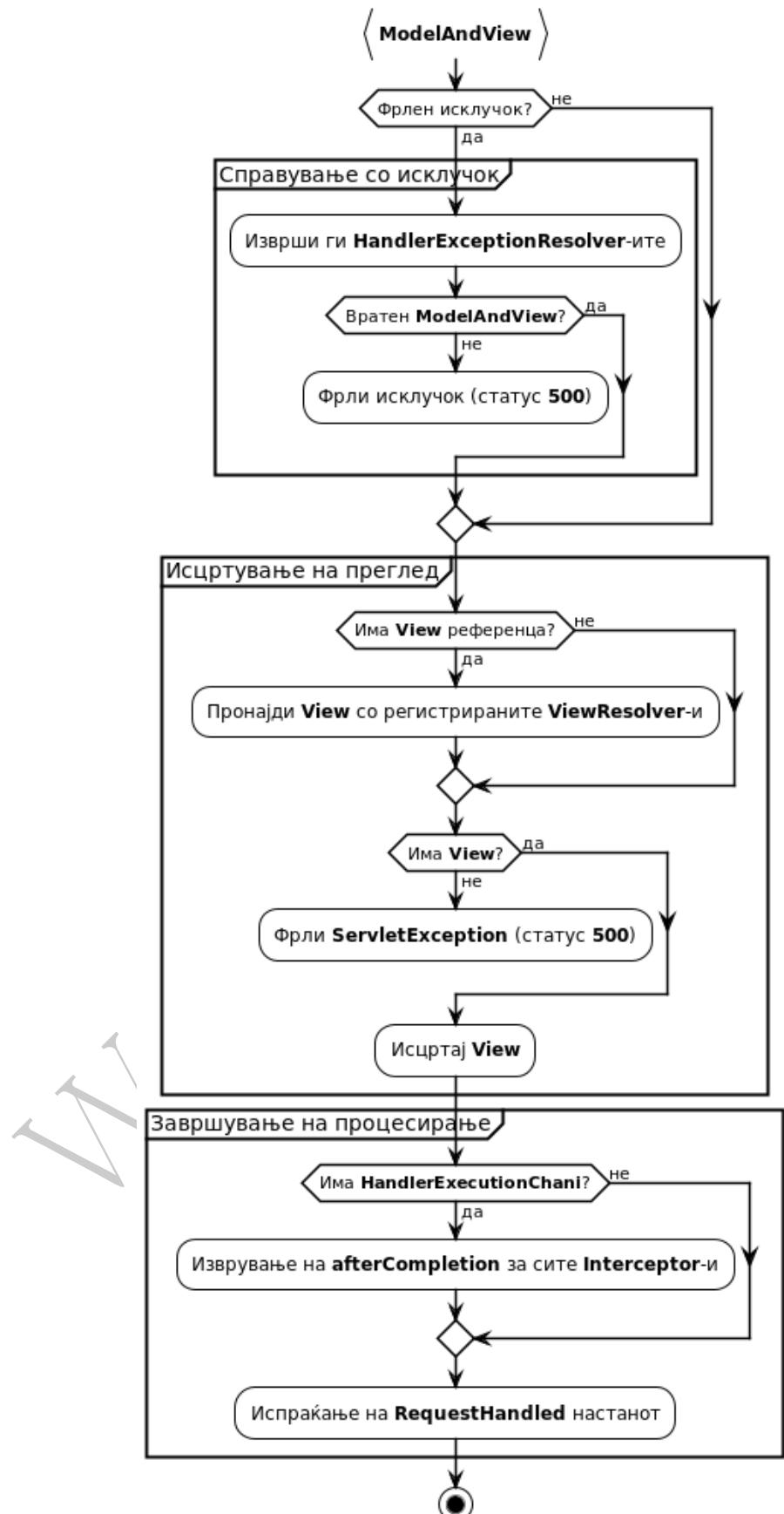


Слика 4-4: Извршување на HandlerExecutionChain

подолу.

Доколку се извршат сите пресретнувачи без да се прекине извршувањето на барањето, компонентата за справување на барањето се делегира на претходно избраниот `HandlerAdapter` кој знае како да го изврши избраниот (`RequestHandler`) и како да го преведе одговорот во `org.springframework.web.servlet.ModelAndView`. Ако нема приказ во вратениот `ModelAndView`, се користи `org.springframework.web.servlet.RequestToViewNameTranslator` за да се генерира име за преглед врз основа на добиеното барање.

Кога ќе се фрли исклучок за време на справувањето со барањето, `DispatcherServlet` ќе се консултира со конфигурираните `org.springframework.web.servlet.HandlerExceptionResolvers` за да се спрavi



Слика 4-5: Справување со исклучоци, исцртување на прегледот и завршување на процесирањето 181

со исклучокот. Имплементацијата на `HandlerExceptionResolvers` може да го преведе исклучокот во приказ за да му покаже на корисникот. На пример, ако постои исклучок поврзан со грешки во базата на податоци, може да се прикаже страница што покажува дека базата на податоци е неисправна. Ако исклучокот не е разрешен, тој повторно се фрла и со него се справува сервлет контејнерот, што генерално резултира со HTTP статусен код за одговор 500 (грешка на внатрешен сервер). Првиот дел од Слика 4-5 го прикажува овој дел од работниот тек за обработка на барања.

Наредниот чекор е исцртување на прегледот. Ако приказот е избран во претходните чекори, `DispatcherServlet` прво проверува дали е референца за поглед (ова е случај ако приказот е `java.lang.String`). Ако е така, конфигурираните `org.springframework.web.servlet.ViewResolvers` се консултираат за да се реши референцата за преглед на вистинска имплементација од `org.springframework.web.servlet.View`. Ако нема преглед и не може да се разреши, се фрла `javax.servlet.ServletException`. Ова е прикажано во вториот дел од Слика 4-5.

Последниот чекор од процесирањето на барањата е завршување на процесирањето. Секое барање поминува низ овој чекор без разлика дали има исклучоци. Ако е достапен `HandlerExecutionChain`, тогаш се повикува методот `afterCompletion` на пресретнувачите. Само пресретнувачите каде методот `preHandle` беше успешно повикан ќе го повикаат нивниот метод `afterCompletion`. Следно, овие пресретнувачи се извршуваат во обратен редослед во однос на редоследот според кој беше повикан нивниот метод `preHandle`. Ова го имитира однесувањето на филтрите за сервлет каде што првиот повикан филтер е и последниот што се повикува. Конечно, `DispatcherServlet` го користи механизмот за настани во Spring Framework за да активира `org.springframework.web.context.support.RequestHandledEvent` (последниот дел од Слика 4-5). Ако има потреба, може да се креира и конфигурира `org.springframework.context.ApplicationListener` за да се примаат и евидентираат овие настани.

4.5 Преглед на функционалностите од Spring MVC модулот

Веб модулот Spring MVC вклучува многу уникатни функции за поддршка на развој на веб апликации:

- Јасна поделба на улогите. Секоја улога (контролер, валидатор, команден објект, форма, модел, `DispatcherServlet`, мапирање на справувањето со ба-

рањата (handler mapping), разрешувач на прегледи (view resolver) и така натаму) може да се исполни со специјализиран објект.

- Моќна и јасна конфигурација на класите од рамката и од апликацијата како JavaBeans. Оваа конфигурациска способност вклучува лесно референцирање низ контекстите, како што се референцирањата од веб-контролерите кон бизнис објекти и валидатори.
- Прилагодливост, ненаметливост и флексибилност. За дадено сценарио, може да се дефинира кој било потпис на методот за справување со барањето што е потребен. За оваа цел може да се користат анотациите за деловите од барањата како што се `@RequestParam`, `@RequestHeader`, `@PathVariable` и други.
- Бизнис логика за повеќекратна употреба, без потреба од дуплирање. Може да се користат постоечките бизнис објекти, како команди или форми, наместо истите да се копираат или да се проширува одредена класа од рамката.
- Приспособливо поврзување (binding) и валидација на аргументите на методите за справување со барањата.
- Приспособливо мапирање на методите на контролерот и разрешување на прегледите. Стратегиите за мапирање на методите на контролерите и на разрешувањето на прегледите се движат од едноставна конфигурација базирана на URL, до софицирани стратегии создадени со специфична намена. Spring е пофлексибилна од веб MVC рамки кои наложуваат одредена конкретна техника.
- Флексиблен пренос на модел. Преносот на модел со мали од име/вредност поддржува лесна интеграција со која било технологија за приказ.
- Приспособлива резолуција на локализација и теми, поддршка за Java Server Pages - JSP со или без библиотеки со специфични тагови од Spring, поддршка за JSTL, поддршка за Velocity без потреба од дополнителни прилагодувања итн.
- Бинови чиј што животен циклус е ограничен на тековното HTTP барање или HTTP сесија. Ова не е специфична карактеристика на самиот Spring MVC, туку на контејнерот(ите) `@WebApplicationContext` што ги користи Spring MVC.

4.6 Пример контролери (Вежби)

Глава 5

Работа со релациони бази на податоци

Секоја софтверска апликација со нејзините корисници разменува податоци кои ги обработува, модифицира, зачувува или чита од некое складиште. Затоа, од големо значење е зачувувањето на податоците во перманентни складишта кои ќе овозможат нивно повторно искористување и по завршувањето на сесијата на корисниците или терминирање на програмата.

Постојат голем број на концепти на зачувување на податоци кои се применуваат при развој на апликациите, но релационите бази на податоци се еден од најстарите и сè уште најчесто користените концепти за апликации за општа намена. Кај релационите бази, податоците се складираат во логички единици наречени табели, за кои можат да се дефинираат меѓусебни релации преку колони назначени како приватни и надворешни клучеви. Манипулацијата на податоците се остварува преку јазикот SQL.

Еден од главните предизвици со кои се соочуваат развивачите на софтвер е големата разлика помеѓу формата на податоците кои ги користат апликациите и формата на ниво складирање во релационите бази на податоци. Кај објектно-ориентираните апликации, податоците се складираат во објекти во меморијата кои содржат својства, а од друга страна, истите тие податоци се запишуваат како една редица во табела на дискот каде секое свойство на објектот претставува колона.

Барањето на податоци од база, иако навидум едноставно, се состои од повеќе чекори кои треба да се имплементираат во кодот на апликацијата, како што се:

- воспоставување на конекција со базата, која неретко може да биде на различна локација,
- испраќање на SQL команди за читање на податоци кои треба да се извршат во базата на податоци,

- преземање на добиените резултати,
- конверзија на добиените податоци од табеларна форма во форма погодна за користење во апликацијата,
- затворање на конекцијата.

Дополнително, апликацијата треба да се справи и со проблеми кои може да се појават кај дел од овие чекори, како што се неможност да се воспостави конекција со серверот, проблем со автентификацијата, грешка во синтаксата на испратената команда и слично.

Имплементацијата на сите овие чекори кај Јава апликациите е овозможена преку библиотеката Java Database Connectivity (JDBC) која претставува стандарден јава апликациски програмски интерфејс (Application Program Interface - API) за комуникација со релациони бази на податоци, независно од нивниот тип. JDBC содржи драјвери (drivers, англ.) за различни релациони бази на податоци (пр. MySQL, PostgreSQL, Oracle, итн.) и преку нив им овозможува на методите од JDBC API да комуницираат со базата на податоци. Иако JDBC ја крие комплексноста на комуникацијата со специфичните релациони бази на податоци, главниот проблем кај овој пристап е што дури и за наједноставно барање за читање на податоци од табела, потребно е да се напишат десетици линии код за сите споменати чекори, вклучувајќи и код за справување со грешки. Сето тоа значително ја зголемува комплексноста при работа со податоци, а оттука и потребното време за развивање на апликации.

Пример за код потребен за селекција на еден продукт со одредена вредност на идентификаторот `id` користејќи JDBC е даден во кодниот сегмент 5.1.

```

1  @Override
2  public Product findById(String id) {
3      Connection connection = null;
4      PreparedStatement statement = null;
5      ResultSet resultSet = null;
6      try {
7          connection = dataSource.getConnection();
8          statement = connection.prepareStatement(
9              "select id, name, price from Product");
10         statement.setString(1, id);
11         resultSet = statement.executeQuery();
12         Product prod = null;
13         if(resultSet.next()) {
14             prod = new Product(
15                 resultSet.getString("id"),

```

```

16                     resultSet.getString("name"),
17                     Decimal.valueOf(resultSet.getString("price")));
18     }
19     return prod;
20 } catch (SQLException e) {
21     // handle SQL Exception
22 } finally {
23     if (resultSet != null) {
24         try {
25             resultSet.close();
26         } catch (SQLException e) {}
27     }
28     if (statement != null) {
29         try {
30             statement.close();
31         } catch (SQLException e) {}
32     }
33     if (connection != null) {
34         try {
35             connection.close();
36         } catch (SQLException e) {}
37     }
38 }
39     return null;
40 }

```

Изворен код 5.1: Пример за имплементација на пребарување на продукт од база на податоци користејќи JDBC API

Освен големиот број линии и комплексноста, при користење на JDBC не постои сепарација помеѓу кодот во SQL и Јава, односно SQL барањата се вметнуваат како текст во самиот код на апликацијата. Ова преставува сериозен проблем доколку се јави потреба да се промени релационата база на податоци, бидејќи различните бази на податоци можат да имаат различни SQL дијалекти, па затоа ќе биде потребно да се пребаруваат SQL командите низ кодот и да се прилагодат на новиот SQL дијалект. Со не помала тежина е и проблемот на лоцирање на грешките во синтаксата на SQL барањата и потребата од повторно компајлирање на целиот Јава код на апликација со цел да се провери исправноста на коригираните барања.

За да се надминат овие проблеми, низ текот на годините се појавуваат голем број на развојни рамки во Јава кои се обидуваат да го поедностават пресликувањето на објекти во редици од табели, да овозможат интероперабилност со различ-

ни релациони бази на податоци и да ја скријат целата комплексност за извршување на барањата кон базите на податоци. Овие рамки главно нудат имплементација на објектно-релационо пресликување ORM (Object-Relational Mapping) и механизми за поедноставена комуникација помеѓу апликацијата и базата на податоци, што значително ја олеснува работата со релациони бази на податоци. Сепак, тие имаат свои недостатоци како што се меѓусебна некомпабилност поради фактот што не се изградени според универзален и отворен стандард или пак нецелосна транспарентност и контрола врз извршувањето која им се дава на развиваите во однос на базата на податоци.

Решението во Јава кое ги адресира сите овие проблеми и нуди едноставен, но истовремено и унифициран начин на пресликување на објекти во релациони бази на податоци, независен од базата на податоци и ORM работната околина е Јава апликацискиот интерфејс за постојаност на податоци - Jakarta Persistence API (JPA). JPA преставува спецификација од типот отворен код за пресликување на објектите и како интерфејс нуди методи за работа со овие објекти, како што се нивно читање, модифицирање, внесување во базата, па дури и извршување на сложени пребарувања напишани во посебен јазик JPQL (Java Persistence Query Language), кој е сличен на SQL. Сепак, JPA не ги имплементира овие функционалности, туку само ги специфицира правилата за пресликување и интерфејсите со методи за работа со објектите кои ќе овозможат нивна размена со базите на податоци. Со оглед на тоа што станува збор само за спецификација, нејзината имплементацијата која би се користела во апликациите треба дополнително да се обезбеди од страна на некој провајдер. Денес постојат повеќе JPA имплементации, но едни од најчесто користените се Hibernate и EclipseLink. Без разлика за која JPA имплементација станува збор, имплементацијата на JPA спецификациите се сведува на генерирање на соодветни SQL наредби и нивно извршување во релациона база на податоци користејќи го JDBC апликацискиот интерфејс и JDBC драјверите. На тој начин, се обезбедува апстракција на работата со базите користејќи објектно-ориентиран пристап, методи за извршување на операциите за креирање, читање, ажурирање и бришење (CRUD - Create, Read, Update, Delete) на објектите како и дополнителни сложени барања кон базата.

И покрај тоа што имплементацијата на JPA крие голем дел од деталите и комплексноста при работа со релациони бази на податоци, развиваите на код повторно треба да креираат посебни репозиториуми за секој пресликан објект во кој ќе ги имплементираат методите за CRUD операции како и посложени-те барања овозможени преку повикување на методите од JPA интерфејсот. За дополнително намалување на комплексноста, големината на кодот, а со тоа и потребното време за работа со релациони податоци се користи апстракцијата Spring Data JPA. Таа е дел од проектот Spring Data на работната рамка Spring,

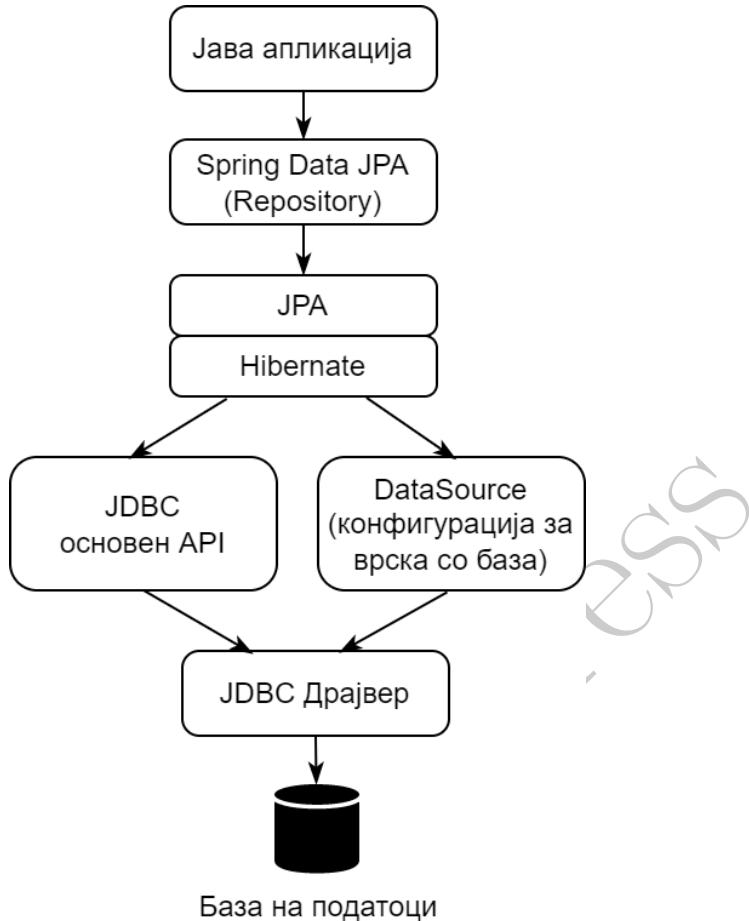
која вклучува различни типови на апстракција за секакви бази на податоци, вклучувајќи ги и нерелационите. Spring Data JPA е главниот тренд за работа со релациони бази на податоци кај развиваите на апликации во Јава бидејќи креира имплементација на сите CRUD функционалности на пресликаните објекти.

На сликата 5-1 е даден преглед на сите нивоа на апстракција кои се користат при работа со релациони бази во Јава. Најдолното ниво е самата база на податоци која прифаќа SQL барања за манипулирања со податоците. За да се овозможи конективност со различни типови на релациони бази на податоци и да се сокријат деталите за целата комуникација помеѓу апликацијата и серверот на бази на податоци, се користат JDBC драјвери. Отворањето на конекција, праќањето на команди кон серверот, нивното извршување, преземање на податоци и затворање на конекцијата се остварува преку JDBC апликацискиот интерфјес. Како дел од ова ниво е и конфигурацијата која му е потребна на драјверот за да се поврзе со базата и кој, меѓу другото, ги содржи адресата на серверот, името на базата и податоците потребни за автеникација со таа база. Следната апстракција е имплементацијата на JPA спецификациите, која во примерот е обезбедена преку JPA имплементацијата Hibernate (предефиниран тип). Оваа имплементација всушност креира код кој ги повикува методите од JDBC API. Во последното, највисоко ниво на апстракција, програмерот само ги дефинира класите за пресликување и репозиториумите, а Spring Data JPA го креира потребниот код за повикување на методите од JPA API кои се имплементирани од страна на JPA провајдерот, на пример имплементацијата Hibernate.

Спецификациите на JPA и Spring Data JPA овозможуваат ефикасно и едноставно развивање на апликации кои работат со релациони бази на податоци. Затоа, целта на оваа глава е да се воведат концептите на Spring Data JPA, спецификациите на JPA за објектно-релационо мапирање и значењето на есенцијалните методи на JPA API.

Иако за развивање на основни апликации методите од JPA API не се неопходни, поради фактот што Spring Data JPA веќе ги имплементира со самата дефиниција на репозиториумите, нивното разбирање е важно за да се разбере концептот на целиот податочен модел, а нивната употреба е од голема полза во понапредни апликации каде е потребна поголема контрола врз извршувањето на пребарувањата од она што го креира Spring Data JPA. За да може да се користи JPA во рамките на спринг проект, во рамките на оваа книга ќе ја користиме зависноста за `spring-data-jpa`, која во `pom.xml` може да се вклучи на следниот начин:

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
```



Слика 5-1: Апстракција на податоци при работа со бази на податоци

```

3   <artifactId>spring-boot-starter-data-jpa</artifactId>
4   </dependency>
5   <dependency>
6     <groupId>com.h2database</groupId>
7     <artifactId>h2</artifactId>
8     <scope>runtime</scope>
9   </dependency>
10  <dependency>
11    <groupId>org.postgresql</groupId>
12    <artifactId>postgresql</artifactId>
13    <scope>runtime</scope>
14  </dependency>
  
```

Изворен код 5.2: Додавање на зависноста за spring-data-jpa во pom.xml

Во изворниот код 5.2 се вклучени и две дополнителни зависности, од кои првата ги вклучува потребните компоненти за поврзување со **h2** базата на по-

датоци која најчесто се користи за тестирање, а втората е **PostgreSql** базата на податоци која е соодветна за продукциска околина. Изборот на база на податоци е надвор од доменот на оваа книга и примерите овде главно ќе се базираат на PostgreSQL, како една од најчесто користените продукциски бази на податоци.

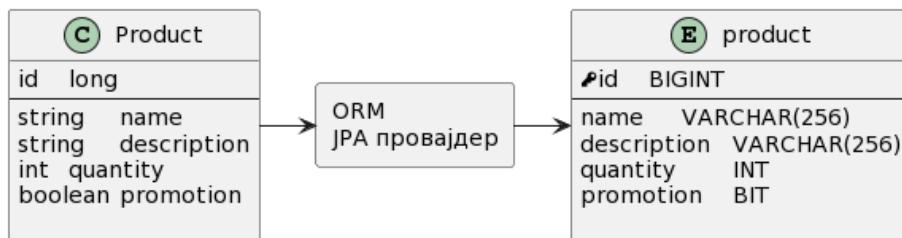
5.1 JPA објектно-релационо пресликување

Еден од клучните елементи на JPA е дефинирање на правила за објектно-ориентирано пресликување ORM. За да се изврши пресликување на една класа во табела во релациона база на податоци, потребно е да се дефинираат правила со мета-податоци кои JPA имплементацијата ќе ги користи за да ги преслика објектите од меморија во редици во одредена табела, и обратно, да ги преслика редиците од табела во објекти во меморија. Постојат два начини на дефинирање на пресликувањата: во посебна XML датотека или преку анотации директно во Јава кодот. Со оглед на тоа што анотациите се поедноставни, поинтуитивни и овозможуваат поголема прегледност на самиот код, нив ќе ги користиме во сите примери во книгата. Карактеристично за JPA е што користи предефинирани правила кои се применуваат во сите случаи освен ако не е посочено поинаку од развишвачот на софтвер. Со тоа се постигнува пресликување со минимални промени на самите класи. За да се користат JPA анотации во Јава кодот, потребно е да се вклучи пакетот: `import jakarta.persistence.*`

Секоја класа која се пресликува во меморија од некоја табела на релациона база се нарекува ентитет (entity, англ.). Која било Јава класа може да се дефинира како ентитет со користење на соодветна анотација. Името на класата се користи за да се определи табелата во базата во која се пресликуваат инстанциите од таа класа, а нејзините својства се користат за да се определат колоните во табелата. Типот на податоци на колоните зависи од типот на податоци на самите својства.

На сликата 5-2 е прикажан едноставен пример на објектно-релационо пресликување на класата `Product` во табелата `product`. Имињата на колоните на табелата се исти како и имињата на својствата. Со оглед на тоа што примитивните и останатите посложени Јава типови на податоци не се исти како и кај релационите бази на податоци, при мапирањето се користат нивните еквиваленти кај конкретната база конфигурирана во JPA имплементацијата.

Секоја инстанца од класата означена како ентитет има можност да биде синхронизирана со базата (`persistable`, англ.), што значи дека може да се стане синхронизирана (зачувана) (`persisted`, англ.). Кога една инстанца е трајна, нејзиниот животен век продолжува и надвор од меморијата. Таа се синхронизира со базата на податоци и која било промена на нејзината состојба, на пример,



Слика 5-2: Објектно-релационо пресликување на класа во табела

вредноста на некое свойство, резултира со промена на соодветната колона во редицата од табелата во базата на податоци во која е пресликана. Синхронизацијата не настапува само при промена на состојбата на инстанцата, туку и при креирање или бришење на инстанца од меморија, што резултира со додавање или бришење на редица од табела во базата на податоци. Сепак, креирање на инстанца од еден ентитет не значи дека таа веднаш ќе се синхронизира со базата бидејќи, сами по себе, новите инстанци не се трајни; тоа се случува дури откако ќе се повика посебен метод од JPA API, за што ќе стане збор подоцна во оваа глава.

Друга важна карактеристика на инстанците од ентитети, неопходна за нивната можност за синхронизација, е секоја инстанца да поседува зачуван идентитет кој се постигнува преку својство кое ќе има улога на уникатен идентификатор исто како што табелите во релационите бази на податоци поседуваат примарен клуч со уникатна вредност. На овој начин се обезбедува еден траен објект да се преслика исклучиво во една редица во табела во базата на податоци.

Ентитетите имаат и својство на атомичност, односно во случај да настане грешка при синхронизацијата во базата после која не може да се продолжи, сите промени што се направиле во базата пред да настане таа грешка нема да бидат зачувани. Атомичноста означува дека или се извршуваат сите промени на ентитетите во базата или не се извршува ниту една.

5.2 Ентитети

Ентитетите преставуваат обични Јава објекти (POJO - Plain Old Java Object) збогатени со анотации кои се користат за дефинирање на пресликувањата. За една Јава класа да се претвори во ентитет, потребно е да се исполнат следниве предуслови:

1. класата да се анотира со `@Entity`,
2. едно од својствата да се анотира со `@Id`,
3. класата да содржи `public` или `protected` конструктор без аргументи.

Овие услови се минималните промени кои треба да се направат во дефиницијата на класата за да се назначи дека класата ќе се пресликува во табела и дека сите нејзини својства ќе претставуваат колони, од кои трајниот идентификатор ќе биде примарен клуч. За сè останато се користат предефинираните поставки, меѓу кои се името на табелата и нејзините колони кои се изведуваат од името на класата и нејзините својства.

Во кодниот сегмент 5.3 е дадена дефиниција на ентитет `Product` според претходно дефинираните услови за ентитети. Класата ја содржи анотацијата `@Data` од пакетот *Lombok*¹ кој се користи за автоматско генерирање на методи на класи.

```

1 import lombok.Data;
2 import javax.persistence.Entity;
3 import javax.persistence.Id;
4 @Data
5 @Entity
6 public class Product {
7     @Id
8     private int id;
9     private String name;
10    private String description;
11    private Double price;
12    private short quantity;
13    private boolean onPromotion;
14    public Product() {}
15 }
```

Изворен код 5.3: Дефиниција на ентитет `Product`

Резултатот од ваквата дефиниција на ентитетот во Spring Boot проект кој користи Hibernate како JPA провајдер и PostgreSQL како база на податоци, ќе биде табелата прикажана на слика 5-3.

Од слика може да се види дека предефинираното пресликување на имињата на табелата и колоните се прави на тој начин што сите букви од името на табелата / својството се мапираат во мали букви, а доколку името е составено од повеќе зборови, според принципот CamelCase (мала почетна буква на првиот збор и голема почетна буква на секој нареден прилепен збор), тогаш сите зборови се претвораат во мали букви и се меѓусебно одвоени со знакот "_" (долна црта). Според тоа, името на ентитетот `Product` се пресликува во име на табела `product`, името на својството `onPromotion` се пресликува во име на колона `on_promotion`.

¹<https://projectlombok.org/>

Сепак, именувањето на табелите и колоните зависи од JPA конфигурацијата и може да се разликува кај различни верзии на Spring Boot.

product	
description	varchar(255)
name	varchar(255)
on_promotion	bit(1)
price	double
quantity	smallint
id	int

Слика 5-3: Табела добиена од пресликување на ентитетот Product

Освен предефинираното пресликување на имиња, може да се користат и кориснички-дефинирани имиња во базата на податоци. Нивното користење е неопходно кога ентитетите или нивните својства имаат имиња кои се резервирали зајмови во одредени бази на податоци, како што се `order`, `user`, `from`, `to`, итн.

5.2.1 Пресликување на табели

За да се специфицира името на табелата во која ќе се преслика ентитетот, се користи анотацијата `@Table` во која се дефинира атрибут `name` со вредност која ќе биде името на табелата.

Во продолжение е даден пример за дефиниција на ентитетот `User`. Во примерот, дефиницијата на ново име на табелата е задолжително бидејќи предефинираното име `user` кое би го креирала JPA имплементацијата ќе предизвика грешка при генерирањето на табелата, поради тоа што `user` е резервиран збор во SQL. Затоа, ентитетот ќе се преслика во табела со име `eshop_user`:

```

1  @Entity
2  @Table(name = "eshop_user")
3  public class User {
4      @Id
5      private int id;
6      private String userName;
7      private String fullName;
8  }

```

5.2.2 Пресликување на колони

За да се промененат предефинираните карактеристики на колоните во базата на податоци, се користи анотацијата `@Column` во која може да се поставуваат следните атрибути:

- `name` за поставување на името на колоната во базата.
- `length` за поставување на должината на колоната во базата. Предефинираната вредност е 255 и најчесто се користи за колони кои содржат низа од знаци или бајти.
- `nullable` за поставување на опцијата колоната да може да содржи празни вредности. Предефинираната вредност е `true`, што значи дека колоната може да содржи вредности `null`, доколку не се обезбеди конкретна вредност за таа колона при внесувањето на податоци.
- `unique` за поставување на опцијата вредностите на колоната да бидат уникатни. Предефинираната вредност е `false`.
- `precision` за поставување на прецизноста, односно вкупниот број на цифри во случај колоната да е од нумерички тип.
- `scale` за поставување на број на цифри после запирката, во случај колоната да се мапира во типот `decimal`. Предефинираната вредност е 0.
- `columnDefinition` за поставување на SQL дефиниција за колоната што ќе се користела при нејзино креирање. Овој параметар нуди можност да се посочи типот на податоци на колона да биде различен од предефинираното мапирање на типови или пак да се постави предефинирана почетна вредност во случај да не се обезбеди од страна на корисникот. Во ова поле може да се вклучат сите претходно набројани поставки како еден SQL коден сегмент. Сепак, користењето на ова свойство може да нè ограничи при промена на база на податоци, затоа што дефиницијата може да не соодветствува на SQL дијалектот на соодветната база.
- `insertable` за поставување на опцијата дали вредностите на колоната да бидат вклучени во SQL наредбата `INSERT`, генерирана од JPA имплементацијата. Предефинираната вредност е `true`.
- `updatable` за поставување на опцијата дали вредностите на колоната да бидат вклучени во SQL наредбата `UPDATE`, генерирана од JPA имплементацијата. Предефинираната вредност е `true`.

Во продолжение е даден пример за редефиниција на претходно разгледуваниот ентитет `Product`, со промена на предефинираните пресликувања на колоните:

```

1  @Entity
2  public class Product {
3      @Id
4      private int id;
5      @Column(name = "product_name", unique = true, nullable = false)
6      private String name;
7      @Column(length = 2000)
8      private String description;
9      @Column(columnDefinition = "decimal(6,2) default 100.00")
10     private Double price;
11     private short quantity = 10;
12     @Column(name = "promotion")
13     private boolean onPromotion;
14 }
```

Според кодот, ќе бидат извршени следните пресликувања на својствата на ентитетот:

- Името на колоната која одговара на својството `name` ќе биде `product_name`. Вредностите мора да се уникантни и различни од `null`.
- Должината на колоната `description` ќе биде 2000 знаци, наместо предефинираните 255.
- Колоната `price` ќе има децимални вредности со вкупно 6 цифри, од кои 2 се децимални цифри². Ако корисникот не обезбеди вредност, тогаш предефинираната вредност која ќе се внесе ќе биде 100.00. Оваа вредност ќе биде внесена од страна на самата база.
- Колоната `quantity` ќе ги задржи предефинираните пресликувања, но ако корисникот не обезбеди вредност, предефинираната вредност која JPA имплементацијата ќе ја предаде на базата ќе биде 10. За разлика од колоната `description`, во овој случај, во дефиницијата на колоната во базата нема да има предефинирана вредност.
- Името на колоната која одговара на својството `onPromotion` ќе биде `promotion`.

5.2.3 Пресликување на примарни клучеви

Според основната дефиниција за ентитети, анотацијата `@Id` се користи за да се специфицира колоната која ќе биде приватен клуч. Како и за секоја колона,

²Истата оваа дефиниција може да се постигне и без `columnDefinition`, доколку се внесе `@Column(precision=6, scale=2)`

така и за колоната за примарен клуч важат истите правила за модифицирање на предефинираните пресликување со помош на анотацијата `@Column` (запазувајќи ги условите за уникатна и различна од `null` вредност).

Анотацијата `@Id` сама по себе не обезбедува механизам за доделување на вредност која ќе ги исполнува условите за уникатност на приватните клучеви, туку само го означува перзистентниот идентитет на ентитетот. Затоа, грижата за генерирање на уникатна вредност останува на програмерот. Во пракса, кога се внесуваат нови ставки во табелите, вредноста на примарниот клуч може да ја генерира JPA имплементацијата или базата на податоци. За да се овозможи оваа функционалност се користи анотацијата `@GeneratedValue`. Постојат четири стратегии за автоматско генерирање на вредност на примарниот клуч. Типот на стратегија се дефинира преку параметарот `strategy` од анотацијата, кој може да добие една од следните вредности:

- `AUTO` означува автоматска селекција на стратегија од страна на JPA имплементацијата врз основа на конфигурираниот SQL дијалект. Ова е предефинирана стратегија, па затоа ако се искористи само анотацијата `@GeneratedValue` без параметри, тогаш се подразбира дека станува збор за автоматска селекција на стратегија. Ако станува збор за имплементацијата Hibernate, тогаш се користи типот `SEQUENCE`.
- `SEQUENCE` означува користење на посебен објект во базата на податоци т.н. секвенца за генерирање на низа од инкрементирачки уникатни броеви, кои потоа JPA имплементацијата ги користи како вредности на примарниот клуч на ентитетот. Неговата употреба зависи од поддршката од страна на базата на податоци. Секвентниот објект е глобален за целата база и може да се референцира преку неговото дефинирано име. Во истата апликација можат да се користат различни секвентни објекти преку дефиниција на различно име. Ако не се специфицира името на секвенцата, тогаш истата секвенца се користи за сите примарни клучеви. Во продолжение е даден пример за користење на стратегија со експлицитно наведување на името на секвентен објект:

```

1  @Id
2  @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
3      "seq_gen")
4  private int id;

```

Овој код ќе генерира секвентен објект во базата со име `"seq_gen"` чија почетна вредност е 1 и секоја следна вредност ќе се зголемува за 50 (предефинирана вредност која може да се промени). За креирање на секвентни објекти со конфигурација различна од предефинираната, се користи анотацијата `@SequenceGenerator`.

тацијата `@SequenceGenerator`, но нејзините детали се надвор од целите на книгата. Оваа стратегија има најдобри перформанси во однос на останатите, бидејќи JPA имплементацијата прави повик до базата за да ја добие иницијалната вредност, а за сите останати вредности до бројот 50 самиот ќе ги генерира вредностите на примарните клучеви без да прави посебни повици до базата. Откако ќе го достигне максималниот број, повторно прави повик до секвентниот објект во базата (SQL наредба `nextval()`) за да ја добие следната вредност (51). Со секвентната стратегија примарниот клуч се доделува од страна на JPA имплементацијата во моментот кога ентитетот ќе стане траен, а повиците до базата се упатуваат само кога ќе биде потребно да се добие следната вредност на секвенцата.

- `IDENTITY` означува користење на колона од типот `identity` и нејзината пријмена исто така зависи од поддршката на базата за ваков тип на колони. Кај колоните од типот `identity` се користи автоматско инкрементирање на вредностите на примарните клучеви. Оваа стратегија има посебни перформанси, особено за истовремено внесување на голем број ентитети, бидејќи вредноста на примарниот клуч ќе се добие дури откако ентитетот ќе биде внесен во базата. Во овој случај, вредноста ја доделува самата база која ѝ ја предава на JPA имплементацијата за да ја постави како примарен клуч на ентитетот.
- `TABLE` означува користење на табела во базата за да се симулира генератор на секвенци. Механизмот е поддржан од сите бази на податоци, па затоа е најфлексибилен во однос на останатите. Секоја редица во табелата го содржи идентификаторот на генераторот и неговата последната генерирана вредност. Кодот во продолжение е пример за користење на стратегија `TABLE`, со конкретно име за генератор:

```

1  @Id
2  @GeneratedValue(strategy = GenerationType.TABLE, generator =
3      "tab_gen")
private int id;

```

За да го добие примарниот клуч кој треба да му го додели на ентитетот, JPA имплементацијата ја чита последната вредност од табелата во база, ја инкрементира и ја ажурира со новодобиената вредност. Очигледно е дека за секој клуч се генерира посебен повик до базата, па затоа стратегијата има најлоши перформанси во однос на другите. Со оглед на тоа што другите механизми се поддржани кај сите познати бази на податоци, оваа стратегија ретко се користи во пракса.

5.2.4 Пресликување на основни типови на податоци

При пресликување на својствата во колони се прави пресликување и на нивните типови на податоци. JPA ги поддржува следните типови на податоци синхронизирачки, односно како типови кои може да ги преслика во база:

- Примитивните типови `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` и `double`.
- Еквивалентните обвиканы класи на примитивните типови од пакетот `java.lang`: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float` и `Double`.
- Низи од знаци и бајти: `char[]`, `byte[]`, `Char[]` и `Byte[]`
- Типови за големи нумерички вредности `java.math.BigInteger`, `java.math.BigDecimal`
- Текстуални низи `java.lang.String`.
- Типови за означување на време во Java `java.util.Date`, `java.util.Calendar`
- Типови за означување на време во JDBC `java.sql.Date`, `java.sql.Time` и `java.sql.Timestamp`.
- Енумерации
- Листи, колекции и мапи
- Серијализабилни класи

За секој од типовите на својствата, JPA имплементацијата одлучува за соодветниот тип во кој ќе се преслика во базата според конфигурираниот SQL дијалект. Од претходните примери може да се воочат следните пресликувања за Postgre база:

- `int` во INTEGER
- `short` во SMALLINT
- `boolean` во BIT
- `String` во VARCHAR(256)

Најголемиот дел од пресликувањата низ различните бази се идентични, без разлика на дијалектот, но постојат типови кои се специфични за одреден дијалект.

5.2.5 Енумерации

Енумерациите во Јава може да се зачуваат во базата, односно да станат трајни, ако пред декларацијата на својството се стави анотацијата `@Enumerated`. Со оглед на тоа што енумерациите претставуваат листа од именувани константи со ordinalни вредности, каде првата има вредност 0 и секоја наредна е за еден поголема од претходната, при пресликувањето на својството во базата на податоци, колоната е од типот INT.

Во продолжение е даден пример каде се дефинира енумерацијата `OrderStatus`:

```
1 public enum OrderStatus {INITIALIZED, PAID, DELIVERED, FINISHED}
```

Вака дефинираната енумерацијата се користи како тип на податок на својството `status` на ентитетот `Order` во примерот во продолжение:

```
1 @Entity
2 @Table(name="eshop_order")
3 public class Order {
4     @Id
5     private int id;
6     private Date date;
7     private double amount;
8     @Enumerated
9     private OrderStatus status;
10 }
```

Според ваквата дефиниција, сите инстанци на кои ќе им се додели вредност `PAID` на својството `status` ќе бидат пресликани во табелата `eshop_order` во редица каде колоната `status` ќе има вредност 1.

Енумерацијата од примерот има само четири елементи, па работата со нумерички вредности не претставува проблем ако се разгледуваат податоците во базата на податоци. Сепак, ваквиот начин на работа може да е несоодветен кога станува збор за побројни енумерации. Доколку ги разгледуваме вредностите во табелата, на прв поглед не би можеле да знаеме за кое име на константа станува збор поради големиот број на ординални вредности. Дополнително, ако се јави потреба да се внесат нови константи во енумерацијата помеѓу веќе постоечките, ординалната вредност на сите константи кои следат после нововнесената константа ќе се зголемат за 1. На пример, ако во претходната енумерација е потребно да се внесе константата `CANCELED` како препоследна во низата, тогаш таа ќе добие вредност 3, а последната константа ќе ја промени својата претходна вредност од 3 во 4. Ова претставува сериозен проблем ако во табелата веќе има вредности бидејќи, после промената, податоците нема да бидат конзистентни и ќе треба да се менуваат (сите нарачки со вредност 3 да се постават на вредност 4) за да одговараат на новата енумерација.

За да се промени предефинирното ординално пресликување (`EnumType.ORDINAL`), како аргумент на енумерацијата може да се пренесе `EnumType.STRING`, со што полето ќе биде пресликано како текстуална колона која, наместо ординални вредности, ќе ги содржи имињата на константите од ену-

мерацијата. Доколку во претходниот пример експлицитно се наведе текстуално мапирање на енумерацијата преку анотацијата `@Enumerated` како во продолжение:

```

1  @Enumerated(EnumType.STRING)
2  private OrderStatus status;

```

тогаш сите инстанци на кои ќе им се додели вредност од енумерацијата, на пример PAID, ќе бидат пресликани во табелата `eshop_order` со текстуалната вредност "PAID" на колоната `status`.

5.2.6 Транзитни полиња

JPA овозможува исклучување на некои од својствата на ентитетите при пресликувањето во колона од табела. На пример, ако е потребно одредено свойство на ентитетот да се искористи за да се зачува привремена пресметана вредност која зависи од други вредности на колоните и не е потребно да зафаќа дополнителен простор во базата, тогаш својството се означува како транзитно со помош на анотацијата `@Transient`. Вредностите на сите транзитни полиња ќе бидат игнорирани од JPA имплементацијата и нема да се испраќаат во базата.

Пример за означување на транзитното свойство `age` со помош на анотација е прикажан во продолжение:

```

1  @Entity
2  public class Customer{
3      @Id private int id;
4      private Date dateOfBirth;
5      @Transient
6      private short age;
7      setAge(){...}
8      ...
9  }

```

Истата функционалност може да се постигне и без анотација, со помош на резервиралиот збор `transient` при декларирање на својството во Јава.

Пример за означување на истото транзитно свойство `age` со помош на резервиран збор е прикажан во продолжение:

```

1  transient private short age;

```

Во овој случај, освен што ќе биде игнорирано од JPA, својството воопшто нема да се серијализира. Ова може да биде проблем доколку објектот треба да

се дистрибуира, на пример, да се испрати од веб сервер до фронтенд на клиент бидејќи при серијализацијата на објектот во JSON, својството ќе биде исклучено. Затоа, препорачливо е транзитните полинја да се означуваат со анотацијата `@Transient`.

5.2.7 Вградени и вградливи објекти

Вградливи објекти се објекти кои се вметнуваат како дел од ентитет, но самите по себе не се ентитети. Еден вградлив објект нема свој перзистилен идентитет и се идентификува со ентитетот во кој е вграден. За еден објект да биде вградлив, при дефиницијата на неговата класа треба да се употреби анотацијата `@Embeddable`, а за истиот да може да се додаде како својство на ентитет потребно е да се употреби анотацијата `@Embedded` при декларација на својството. Својствата на вградливиот објект и на ентитетот во кој тој се вградува се пресликуваат во колони во една иста tabela. Користењето на вградливи објекти овозможува логичка поделба на табелите и повторно искористување на вградливите објекти во други ентитети.

Во изворниот код 5.6 е дадена дефиницијата на ентитетот `Customer` чии својства поврзани со адресата (`street`, `city`, `postalCode` и `country`) треба да се поделат во посебна логичка единица, но сепак, нивните колони да бидат дел од истата tabela во која ќе се пресликаат и останатите колони.

```

1  @Entity
2  public class Customer {
3      @Id
4      @GeneratedValue
5      private int id;
6      private String firstName;
7      private String lastName;
8      private String street;
9      private Integer number;
10     private String city;
11     private Integer postalCode;
12     private String country;
13 }
```

Изворен код 5.4: Пример за ентитет со својства кои треба да се поделат во посебни логички единици

Зададената цел за логичка поделба на својствата од `Customer` се постигнува со групирање на својствата за адреса во посебна вградлива класа `Address`, чија дефиниција е дадена во изворниот код 5.5.

```

1  @Embeddable
2  public class Address {
3      private String street;
4      private String city;
5      private Integer postalCode;
6      private String country;
7  }

```

Изворен код 5.5: Пример за вградлива класа

Во изворниот код 5.6 е дадена дефиницијата на ентитетот `Customer` со вградено свойство кое ја користи вградливата класа `Address`.

```

1  @Entity
2  public class Customer {
3      @Id
4      @GeneratedValue
5      private int id;
6      private String firstName;
7      private String lastName;
8      @Embedded
9      private Address address;
10 }

```

Изворен код 5.6: Пример за ентитет со вградена класа

Резултатот од пресликувањето на ентитетот `Customer` со вградена класа ќе биде табелата `customer` прикажана на слика 5-4.

customer	
	city varchar(255)
	country varchar(255)
	first_name varchar(255)
	last_name varchar(255)
	postal_code int
	street varchar(255)
	id int

Слика 5-4: Табела добиена од пресликување на ентитетот `Product`

Од сликата може да се забележи дека сите колони на табелата се добиени со пресликување на својствата на ентитетот `Customer` и на вградената класа `Address`. Од аспект на база на податоци, ваквата логичка поделба на класи нема никакво значење бидејќи станува збор за една табела која би се добила на идентичен начин како и кога би постоел само ентитет кој директно би ги содржал својствата на вградливата класа. Сепак, од објектно-ориентиран аспект, поделбата има предности како што се:

- логичко груирање на различни својства, што овозможува поголема прегледност и нивно поедноставно користење особено кога станува збор за ентитети со голем број на својства,
- можност за повторно искористување, што значи дека истата класа може да се користи како вградена класа и во други ентитети, без потреба да се декларираат истите својства во секој посебно.

Во случај на повторно искористување на една иста вградлива класа во различни ентитети, за секој ентитет се креира посебна табела која ги содржи сите пресликани колони од вградливата класа. Но, што доколку едно својство од вградливата класа која се користи во повеќе ентитети треба да се преслика во колона со различно име? Во тој случај, веднаш по анотацијата `@Embedded` при декларацијата на вградливото својство на ентитетот, се додадава анотацијата `@AttributeOverrides` со чија помош се редефинира пресликувањето на избрани својства од вградливата класа во колони во табелата која одговара на самиот ентитет. Како аргумент на анотацијата се пренесува листа од анотации `@AttributeOverride`. За секој елемент од листата на анотации се пренесуваат параметрите:

- `name` за да се назначи својството кое се редефинира и
- `column` за да се посочи името на колоната во кое ќе се преслика својството, како и други поставки за колоната овозможени преку анотацијата `@Column`.

Во изворниот код 5.7 е даден пример за ентитетот `Company` кој ја користи истата вградлива класа `Address` од изворниот код 5.5, но со редефиниција на пресликувањата на нејзините својства `street` и `postalCode`.

```

1  @Entity
2  public class Company {
3      @Id @GeneratedValue
4      private int id;
5      private String name;
```

```

6   private String taxNumber;
7   @Embedded
8   @AttributeOverrides({
9       @AttributeOverride(name="street",
10      column=@Column(name="street_name")),
11      @AttributeOverride(name="postalCode",
12      column=@Column(name="zip_code"))
13  })
14  private Address address;
15 }
```

Изворен код 5.7: Пример за ентитет со вградена класа со редефинирани својства

Резултатот од ваквото пресликување ќе биде табелата `company` прикажана на слика 5-5.

company	
city	varchar(255)
country	varchar(255)
number	int
zip_code	int
street_name	varchar(255)
name	varchar(255)
tax_number	varchar(255)
 id	int

Слика 5-5: Табела добиена од пресликување на ентитетот Company со редефиниција на својства на вградлива класа Address

Од сликата може да се забележи дека својствата `street` и `postalCode` се пресликани во колоните `street_name` и `zip_code`, соодветно, за разлика од слика 5-4 каде тие предефинирано се пресликуваат во колоните `street` и `postal_code`.

5.3 Релации

Ентитетите кои ги разгледавме до сега се дефинирани како изолирани класи со примитивни својства или вградливи класи кои се пресликуваат во посебни табели без меѓусебна зависност. Меѓутоа, во реалните апликации, постојат релации помеѓу ентитетите, односно еден ентитет може да има референци кон еден или

повеќе други ентитети, но исто така, и другите ентитети можат да имаат врски кон тој ентитет.

На пример, ако една нарачка е претставена со ентитетот `Order` и таа нарачка е направена од купувач претставен со ентитетот `Customer`, тогаш постои релација помеѓу тие два ентитета во насока од изворниот ентитет `Order` кон дестинацискиот ентитет `Customer`. Во објектно-ориентираниот модел, релацијата може едноставно да се имплементира ако на нарачката ѝ се додаде референца кон купувачот. Меѓутоа, можат да постојат повеќе различни ентитети `Order` кои се насочени кон истиот ентитет `Customer`, во случај кога повеќе нарачки се направени од еден ист купувач. Затоа, за оваа врска се вели дека е од типот повеќе-кон-еден (*many-to-one*, англ.).

Освен во насоката нарачка-купувач, постои и релација во насока купувач-нарачка. Во овој случај, извор на релацијата е купувачот, односно ентитетот `Customer`, но со оглед на тоа што тој може да направи повеќе различни нарачки, оваа релација ќе има повеќе дестинациски ентитети `Order`. Затоа, релацијата во насока од `Customer` кон `Order` е од типот еден-кон-повеќе (*one-to-many*, англ.).

Имплементацијата на оваа насока од релацијата во објектно-ориентираниот модел се постигнува доколку на купувачот му се додадат референци кон сите нарачки кои ги направил, или во пракса, ако на купувачот му се додаде референца кон колекција која ќе ги содржи сите нарачки.

Според тоа, може да се заклучи дека врските меѓу ентитетите имаат различни насоки и може да бидат од различен тип. Во примерот со ентитетите `Order` и `Customer`, може да постои референца во класата `Order` кон `Customer`, но да не постои колекција од ентитети `Order` во `Customer`. Во тој случај, `Order` „знае“ за постоењето на `Customer`, но `Customer` „не знае“ за постоењето на колекцијата од ентитети `Order`. Другата опција е ако и кај двата ентитети се дефинирани меѓусебни референци, односно ако и ентитетот `Customer` „знае“ за постоењето на сите ентитети `Order`.

Според типот, односно бројноста на извornите и дестинациските ентитети, релациите можат да бидат:

- *Еден-кон-еден* (*one-to-one*, англ.)
- *Повеќе-кон-еден* (*many-to-one*, англ.)
- *Еден-кон-повеќе* (*one-to-many*, англ.) и
- *Повеќе-кон-повеќе* (*many-to-many*, англ.).

За разлика од објектно-ориентираниот модел каде релациите едноставно се имплементираат со референци кон други поединечни ентитети или пак колекција од ентитети, кај релационите бази на податоци, релациите помеѓу табелите

добиени со пресликување на ентитетите се имплементираат со користење на надворешни клучеви (foreign keys, англ.) и дополнителни табели. Кај базите на податоци не постојат насоки на релациите и секогаш можат да се добијат податоци од колоните на табелите од двете страни на релацијата. Со цел да се надмине ваквата концептуална разлика помеѓу објектно-ориентираниот модел и моделот на бази на податоци, JPA обезбедува анотации за означување на својствата на ентитетите, кои сами по себе претставуваат ентитети или коллекции од ентитети. Преку анотациите, JPA имплементацијата го генерира соодветниот SQL код за имплементација на релациите во релационите бази на податоци. Во продолжение ќе го разгледаме користењето на JPA анотациите за сите типови на релации помеѓу ентитетите.

5.3.1 Релација „еден-кон-еден“

Кај релацијата еден-кон-еден изворниот ентитет е во релација само со еден дестинациски ентитет. Релацијата се користи кога е потребно својствата на еден ентитет да се поделат и логички на ниво на класа, но и физички на ниво на база на податоци. Физичката поделба означува дека својствата ќе бидат пресликани во колони во различни табели. Бидејќи станува збор за различни табели, тогаш се работи за два различни ентитети кои се поврзани преку релација еден-кон-еден. Кај релацијата еден-кон-еден една инстанца од изворниот ентитет може да е поврзана само со една инстанца на дестинацискиот ентитет, односно не е можно споделување на иста инстанца од дестинацискиот ентитет од страна на повеќе инстанци на изворниот ентитет.

За дефинирање на релација еден-кон-еден во JPA се користи анотацијата `@OneToOne` во изворниот ентитет, пред декларацијата на својството од типот на дестинацискиот ентитет.

Да ја земеме за пример класата `Customer` дадена во изворниот код 5.4. За да се подели на два ентитети кои ќе бидат пресликани во различни табели, а сепак да бидат поврзани, својствата поврзани со адресата ќе бидат дефинирани во посебна класа `Address`, но овој пат таа класа ќе биде декларирана како ентитет. Дефиницијата на ентитетот `Address` е прикажана во изворниот код 5.8. Може да се забележи дека за да биде ентитет, класата мора да поседува и перзистентен идентитет, па затоа е додадено својството `id`, анотирано со `@Id`.

```

1 @Entity
2 public class Address {
3     @Id @GeneratedValue
4     private int id;

```

```

5   private String street;
6   private Integer Number;
7   private String city;
8   private Integer postalCode;
9   private String country;
10 }

```

Изворен код 5.8: Пример за ентитет кој е дел од еден-кон-еден релација

Во изворниот код 5.9 е дадена дефиницијата на ентитетот `Customer` која има врска еден-кон-еден со дестинацискиот ентитет `Address`. Во дефиницијата за класата е додадена анотацијата `@OneToOne` пред декларацијата на својството `address`, преку кое се остварува таа релација.

```

1 @Entity
2 public class Customer {
3     @Id @GeneratedValue
4     private int id;
5     private String firstName;
6     private String lastName;
7     @OneToOne
8     private Address address;
9 }

```

Изворен код 5.9: Пример за ентитет со врска еден-кон-еден

Според ваквата дефиниција на класите ќе се генерираат табелите прикажани на слика 5-6.



Слика 5-6: Табели добиени од пресликување на ентитетите `Customer` и `Address` со врска еден-кон-еден

На сликата 5-6 се гледа дека имплементацијата на врската еден-кон-еден е постигната преку надворешниот клуч `address_id` во табелата `customer` кој е вклучен во табелата `address`. Според JPA имплементацијата, името

на колоната што се додава како надворешен клуч се добива кога на името на својството `address` во изворниот ентитет ќе се додаде долна црта ("_") и името на колоната во која се пресликува својството со улога на примарен клуч во дестинацискиот ентитет `id`. Но, што доколку табелите се веќе креирани и надворешниот клуч постои со име различно од предефинираното? Во тој случај, се користи анотацијата `@JoinColumn` во која преку параметарот `name` се поставува името на надворешниот клуч.

Во примерот за ентитетот `Customer`, името на надворешниот клуч може да се постави да биде `addr_id` преку следниот код:

```

1  @OneToOne
2  @JoinColumn(name="addr_id")
3  private Address address;

```

Позицијата на надворешниот клуч во табелите на ентитетите дефинира уште една важна класификација на страните во релациите, независно од тоа кој ентитет е изворен, а кој е дестинациски:

- *сопственик* на релацијата (*owner*, англ.) - ентитетот чија пресликана табела го поседува надворешниот клуч и
- *инверзна* (*inverse*, англ.) или *поседувана* (*owned*, англ.) страна - другата страна од релацијата.

Во релацијата еден-кон-еден во примерот, ентитетот `Customer` се смета за *сопственик*, а ентитетот `Address` се смета за *поседувана страна*. Анотацијата `@JoinColumn` секогаш се дефинира кај ентитетот-сопственик.

Во релацијата во примерот, во објектно-ориентираниот модел не постои референца од `Address` кон `Customer`. Во случај да биде потребна референца и во обратната насока, тогаш се дефинира **двонасочна** еден-кон-еден релација според следниве правила:

- Кај ентитетот сопственик на релацијата, пред својството кое ја дефинира релацијата кон инверзниот ентитет се поставува анотацијата `@OneToOne`, а доколку има потреба за експлицитно именување на надворешниот клуч се додава анотацијата `@JoinColumn` во која преку параметарот `name` се поставува името на надворешниот клуч (втората нотација не е неопходна, но е препорачливо да се употребува за да се означи кој е ентитет сопственик).
- Кај инверзната страна на релацијата, пред својството кое ја дефинира релацијата кон ентитетот-сопственик се поставува анотацијата `@OneToOne` и преку параметарот `mappedBy` се поставува името на својството од ентитетот-сопственик кое го референцира инверзниот ентитет.

Ако се употребат овие правила за да се дефинира двонасочна еден-кон-еден релација помеѓу ентитетите `Customer` и `Address`, се добива кодот прикажан во кодните сегменти 5.10 и 5.11.

```
1 @Entity
2 public class Customer {
3     @Id @GeneratedValue
4     private int id;
5     private String firstName;
6     private String lastName;
7     @OneToOne
8     private Address address;
9 }
```

Изворен код 5.10: Пример за ентитет-сопственик на двонасочна еден-кон-еден релација

```
1 @Entity
2 public class Address {
3     @Id @GeneratedValue
4     private int id;
5     private String street;
6     private String city;
7     private Integer postalCode;
8     private String country;
9     @OneToOne(mappedBy="address")
10    private Customer customer;
11 }
```

Изворен код 5.11: Пример за инверзен ентитет од двонасочна еден-кон-еден релација

Резултатот од вака анотираните ентитети ќе бидат истите табели како и на слика 5-6, но во објектно-ориентираниот модел и ентитетот `Address` ќе има пристап до својствата на ентитетот `Customer`.

Поради специфичноста на примерот, во релацијата еден-кон-еден ентитетот `Customer` беше сопственик на релацијата. Сепак, релацијата може да се дефинира и на начин при кој `Address` ќе биде сопственик, односно неговата пресликана табела ќе го содржи надворешниот клуч. Тоа може да се постигне доколку кај `Address` се направи следната промена:

```

1  @Entity
2  public class Address {
3      ...
4      @OneToOne
5      private Customer customer;
6  }

```

Додека пак, кај ентитетот `Customer` се направи следната промена за да тој стане инверзен ентитет:

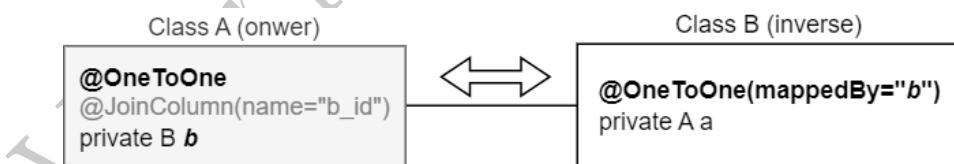
```

1  @Entity
2  public class Customer {
3      ...
4      @OneToOne(mappedBy="customer")
5      private Address address;
6  }

```

Доколу се пропушти `mappedBy` во анатацијата кај инверзниот ентитет, тогаш JPA имплементацијата нема да знае кој е сопственик на релацијата, па затоа и во двете табели ќе се додадат колони со надворешни клучеви. Таквата ситуација е идентична како и кога би се дефинирале две посебни еднонасочни релации. Затоа, при дефиниција на двонасочни релации секогаш треба да се употреби `mappedBy` во рамките на анатацијата `@OneToOne` и тоа само кај инверзниот ентитет.

За поедноставна интерпретација на правилата за дефинирање на релација еден-кон-еден, на слика 5-7 е даден шематски приказ на два ентитети заедно со анатациите за дефинирање на релацијата. На сликата задолжителните анатации се задебелени, а опционалните се означени со посветла боја.



Слика 5-7: Шематски приказ на дефиниција за двонасочна релација еден-кон-еден

Доколку не е потребна двонасочна релација, тогаш се користи само анатацијата `@OneToOne` без аргумент кај сопственикот на релацијата.

5.3.2 Релација „повеќе-кон-еден“

Релацијата повеќе-кон-еден има сличен концепт и дефиниција како и еден-кон-еден, но се разликува по тоа што повеќе инстанци од изворниот ентитет се во релација само со една инстанца од дестинацискиот ентитет. Оваа релација е една од најчесто користените. Таа се користи кога е потребно да се направи асоцијација кон некоја инстанца од дестинациски ентитет без ограничување дали некоја друга инстанца од изворниот ентитет веќе има направено таква асоцијација. Каде релацијата повеќе-кон-еден една инстанца на дестинацискиот ентитет може да биде споделна од повеќе изворни ентитети.

За дефинирање на релација повеќе-кон-еден во JPA се користи анотацијата `@ManyToOne` и тоа кај изворниот ентитет, пред декларацијата на својството од типот на дестинацискиот ентитет.

Како пример за повеќе-кон-еден врска ќе ја земеме релацијата помеѓу ентитетите `Order` и `Customer` кои ги разгледавме во воведниот дел за релации. За потсетување, повеќе различни нарачки можат да бидат направени од еден ист купувач. Во оваа релација, `Order` е изворен ентитет и неговата дефиницијата е прикажана во изворниот код 5.12. Дефиницијата на `Customer` е иста како и претходно, но е повторно прикажана во кодот 5.13, игнорирајќи ја релацијата еден-кон-еден со ентитетот `Address`.

```

1  @Entity
2  @Table(name="eshop_order")
3  public class Order {
4      @Id @GeneratedValue
5      private int id;
6      private Date date;
7      private double amount;
8      private Status status;
9      @ManyToOne
10     private Customer customer;
11 }
```

Изворен код 5.12: Пример за изворен ентитет на релација повеќе-кон-еден

```

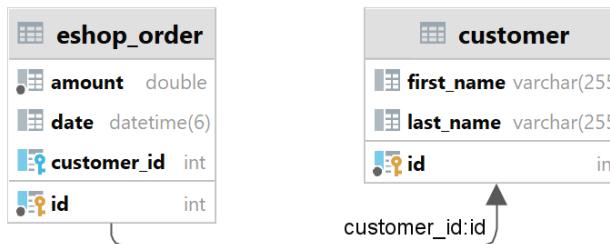
1  @Entity
2  public class Customer {
3      @Id @GeneratedValue
4      private int id;
5      private String firstName;
```

```

6   private String lastName;
7   ...
8 }
```

Изворен код 5.13: Пример за дестинациски ентитет на релација повеќе-кон-еден

Ентитетите заедно со релацијата повеќе-кон-еден од примерот ќе се пресликаат во табелите прикажани на слика 5-8.



Слика 5-8: Табели добиени од пресликување на ентитетите Order и Customer со врска повеќе-кон-еден

Исто како и кај врската еден-кон-еден и во овој случај имплементацијата на релацијата е постигната преку надворешниот клуч `customer_id` во табелата `eshop_order`. Предефинираното име на надворешниот клуч е `customer_id` добиено според правилата за именување, но истото може да се промени со анотацијата `@JoinColumn`.

5.3.3 Релација „еден-кон-повеќе“

Релацијата еден-кон-повеќе претставува релација која вклучува колекција од ентитети. Кај неа, една инстанца од изворниот ентитет има референца кон повеќе инстанци од дестинацискиот ентитет. Референцата кон повеќе инстанци кај изворниот ентитет се имплементира со користење на колекции.

За дефинирање на еднонасочна релација еден-кон-повеќе, во JPA се користи анотацијата `OneToMany` во изворниот ентитет пред декларацијата на својството од типот на дестинацискиот ентитет.

Како пример за врската еден-кон-повеќе ќе ги земеме истите ентитети `Order` и `Customer`, но во овој случај, ќе ја разгледаме релацијата во насока од `Customer` кон `Order`. Дефиницијата на изворниот ентитет `Customer` е дадена во изворниот код 5.14. Имплементацијата на релацијата од една инстанца на `Customer` кон повеќе различни инстанци на `Order` се постигнува преку својството `orders` декларирано како листа од `Order`. При декларацијата се креира инстанца од празна листа `ArrayList<>` која може едноставно да се пополни и уредува преку

нејзините методи. Овој начин на декларација со иницијализација има предност што својството `orders` има вредност различна од `null`, па во случај инстанцата `Customer` да нема поврзани ентитети од типот `Order`, повикувањето на нејзиниот метод `add()` нема да резултира со грешка како резултати на повик на метод од `null` вредност која би се исфрлила во случај ако листата се декларира без да се иницијализира.

```

1  @Entity
2  public class Customer {
3      @Id @GeneratedValue
4      private int id;
5      private String firstName;
6      private String lastName;
7      @OneToMany
8      private List<Order> orders = new ArrayList<>();
9  }

```

Изворен код 5.14: Пример за изворен ентитет на еднонасочна релација еден-кон-повеќе

Дефиницијата на дестинацискиот ентитет `Order` е дадена во изворниот код 5.15. Кај оваа дефиниција е отстрането својството `customer`, со цел врската меѓу ентитетите да биде еднонасочна.

```

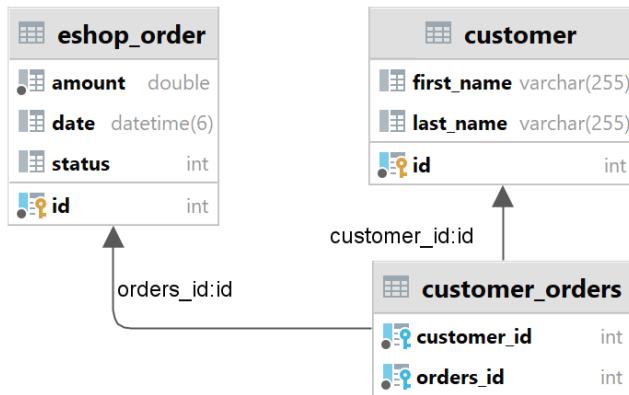
1  @Entity
2  public class Order {
3      @Id @GeneratedValue
4      private int id;
5      private Date date;
6      private double amount;
7      private Status status;
8  }

```

Изворен код 5.15: Пример за дестинациски ентитет на релација еден-кон-повеќе

Ентитетите заедно со релацијата еден-кон-повеќе од примерот ќе се пресликаат во табелите прикажани на слика 5-9.

Од сликата може да се забележи дека освен пресликаните табели `customer` и `eshop_order`, генерирана е и трета tabela `customer_order` која се состои од два надворешни клучка, по еден од двете табели од ентитетите. Причината за ова е неможноста на JPA имплементацијата да одлучи каде да ја смести колоната за надворешен клуч. Кај ваквата релација, ентитетот `Order` е сопственик на ре-



Слика 5-9: Табели добиени од пресликување на ентитетите Customer и Order со еднонасочна врска еден-кон-повеќе

лацијата, па затоа треба да го содржи надворешниот клуч. Но, во овој случај, бидејќи кај сопственикот нема анотација `@ManyToOne`, имплементацијата не може да знае каде да ги смести сите надворешни клучеви. Користењето на третата табела троши дополнителни ресурси и во базата, но и во меморијата, па затоа не е препорачливо решение. Еден-кон-многу релација генерално не е препорачливо да се користи поради лоши перформанси.

За да се избегнат ваквите недостатоци, се користи **двонасочна** релација која, покрај еден-кон-повеќе, опфаќа и врска повеќе-кон-еден. За дефинирање на ваква релација потребно е да се следат следните правилна:

- Се додава анотацијата `@OneToMany` во инверзниот ентитет пред декларацијата на својството од типот колекција од ентитет сопственик. Во анотацијата се пренесува и аргументот `mappedBy`, чија вредност е името на својството преку кое се дефинира релацијата во ентитетот сопственик. Оваа страна од релацијата е инверзна.
- Се додава анотацијата `@ManyToOne` во ентитетот сопственик пред декларацијата на својството од типот на инверзниот ентитет. Може да се користи анотацијата `@JoinColumn` за да се означи дека оваа страна е сопственик на релацијата.

Ако се водиме според оваа дефиниција за двонасочна релација еден-кон-повеќе помеѓу класите `Customer` и `Order`, тогаш првото правило ќе се примени на ентитетот `Customer` и неговата дефиниција ќе биде како во кодниот сегмент 5.16. Со примена на второто правило на ентитетот `Order` ќе се добие идентична дефиниција како и кодниот сегмент 5.12.

```
1 @Entity
2 public class Customer {
3     @Id @GeneratedValue
4     private int id;
5     private String firstName;
6     private String lastName;
7     @OneToMany(mappedBy="customer")
8     private List<Order> orders
9 }
```

Изворен код 5.16: Пример за дестинациски ентитет на двонасочна релација еден-кон-повеќе

Табелите добиени од пресликувањето на ентитетите `Order` и `Customer` со двонасочна еден-кон-повеќе (или повеќе-кон-еден) релација, се идентични како табелите прикажани на слика 5-8.

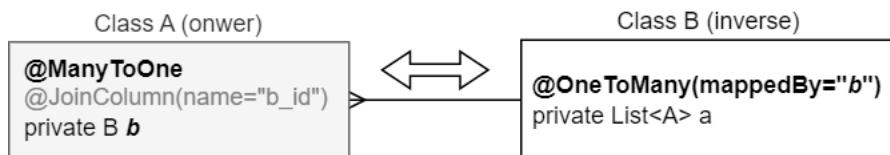
При дефиниција на релација еден-кон-повеќе, анотацијата `@OneToMany` секогаш се назначува кај својство кое е **колекција** од ентитети. Во конкретните примери беше користена колекција од типот `List`, но во зависност од потребата, може да се користат и колекции од типот `Set` и `Map`. Главната карактеристика на последните две колекции е фактот дека тие не дозволуваат дупликати и елементите не можат да се подредат, па затоа се поограничуваат во однос на `List`. Дополнителна предност на `List` е фактот дека може да се наведе својството според кое ќе се подредат елементите при нивно читање од базата во меморијата, преку анотацијата `@OrderBy`. Како аргумент на оваа анотација се пренесува името на својството според кое ќе се подредат елементите и насоката на подредување. Предефинираното својство за подредување е перзистентниот идентификатор (својството анотирано со `@Id`), а предефинирана насока е растечка (`ASC` (ascending, англ.)), што значи дека доколку не е наведено поинаку, елементите ќе бидат подредени според редоследот на внесување во базата.

За да се подредат нарачките по опаѓачки редослед на својството `amount` во листата од кодниот сегмент 5.16, потребно е да се направи следната промена на својството `orders`:

```
1 @OneToMany(mappedBy="customer")
2     @OrderBy("amount DESC")
3     private List<Order> orders
```

Правилата за дефиниција на двонасочна релација еден-кон-повеќе се шематски прикажани на слика 5-10.

Правилата од дијаграмот можат да се искористат и за дефинирање на врска



Слика 5-10: Шематски приказ на дефиниција за еднонасочна и двонасочна релација еден-кон-повеќе

повеќе-кон-еден и за врска еден-кон-повеќе, поединечно.

5.3.4 Релација „повеќе-кон-повеќе“

Релацијата повеќе-кон-повеќе, слично како и еден-кон-повеќе, припаѓа на групата релации кои вклучуваат колекција од ентитети. Кај неа, една инстанца од изворниот ентитет има референца кон повеќе инстанци од дестинацискиот ентитет, но и обратно, една инстанца од дестинацискиот ентитет содржи повеќе инстанци од изворниот ентитет. Референците кон повеќе инстанци и кај изворниот и кај дестинацискот ентитет се имплементираат со користење на колекции.

За дефинирање на еднонасочна релација повеќе-кон-повеќе се користи анотацијата `@ManyToMany` во изворниот ентитет пред декларацијата на својството од типот листа од дестинациски ентитети.

Како пример за врската повеќе-кон-повеќе ќе ги земеме ентитетите `Product` и `Category` и ќе ја разгледаме насоката од `Product` кон `Category`. Еден продукт може да припаѓа на повеќе разни категории, односно една инстанца од `Product` може да референцира повеќе инстанци од типот `Category`, па затоа во него ќе се декларира својство `categories` кое е листа од ентитети од типот `Category`. Од друга страна, една категорија може да содржи повеќе продукти, односно една инстанца од `Category` може да референцира повеќе инстанци од типот `Product` кои би се претставиле како листа од ентитети од типот `Product`, но оваа листа не се декларира како својство на класата `Category` бидејќи станува збор за еднонасочна повеќе-кон-повеќе врска. Дефиницијата на овие две класи е прикажана во изврните кодови 5.17 и 5.18.

```

1 @Entity
2 public class Product {
3     @Id @GeneratedValue
4     private int id;
5     private String name;
6     private String description;
7     private Double price;

```

```

8     @ManyToMany
9     private List<Category> categories;
10 }

```

Изворен код 5.17: Пример за изворен ентитет на еднонасочна релација повеќекон-повеќе

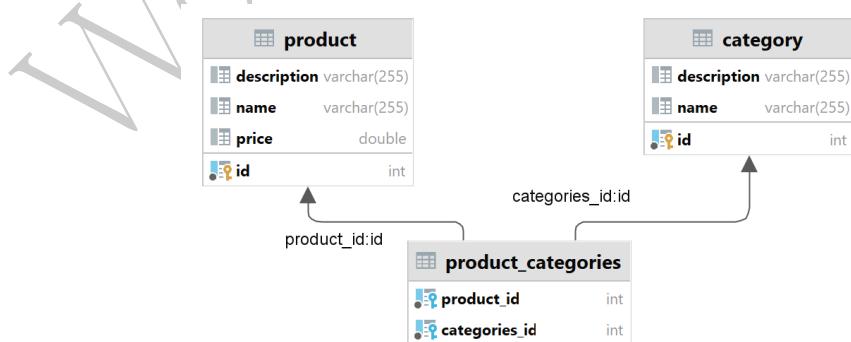
```

1  @Entity
2  public class Category {
3      @Id @GeneratedValue
4      private int id;
5      private String name;
6      private String description;
7  }

```

Изворен код 5.18: Пример за дестинациски ентитет на релација повеќекон-повеќе

Ентитетите, заедно со релацијата повеќекон-повеќе од примерот, ќе се пресликаат во табелите прикажани на слика 5-11. Според slikата, освен пресликаните табели `product` и `category`, генерирана е и трета табела, `product_categories`, која се состои од два надворешни клучка, по еден од двете табели од ентитетите. За разлика од врската еден-кон-повеќе каде постоише начин да се избегне користење на трета табела, кај врската повеќекон-повеќе третата табела е единствен начин за нејзина имплементација во базата на податоци. Причината за тоа е што и во двете табели на ентитетите во секоја редица потребно е да се зачува листа од надворешни клучеви кон другата табела, што се постигнува со нова табела каде секоја редица е составена од два надворешни клучка, што овозможува дефиниција на која било комбинација на бројност од двета ентитети.



Слика 5-11: Табели добиени од пресликување на ентитетите `Product` и `Category` со еднонасочна врска повеќекон-повеќе

Во еднонасочната релација, ентитет сопственик е `Product`, иако самата табела не содржи надворешен клуч, а инверзен ентитет е `Category`. За да се препокрие предефинираното именување на третата табела која се користи за спојување на двете табели (join table, англ.) и нејзините колони (join columns, англ.), веднаш по анотацијата `@ManyToMany` се користи анотацијата `@JoinTable` во која преку параметарот `name` се дефинира името на табелата, преку параметарот `joinColumns` се дефинира колоната за надворешниот клуч кој се однесува на сопственикот на релацијата, а преку `inverseJoinColumns` се дефинира колоната за надворешниот клуч кој се однесува на инверзната страна од релацијата. За специфицирање на двете колони се користи веќе познатата анотација `@JoinColumn`.

Во изворниот код 5.19 е прикажано користењето на анотацијата `@JoinTable` со чија помош табелата за спојување се именува како `product_category`, а колоните за нејзините надворешни клучеви како `prod_id` и `cat_id`.

```

1  @Entity
2  public class Product {
3      @Id @GeneratedValue
4      private String name;
5      private String description;
6      private Double price;
7      @ManyToMany
8      @JoinTable(name = "product_category",
9          joinColumns = @JoinColumn(name = "prod_id"),
10         inverseJoinColumns = @JoinColumn(name = "cat_id"))
11      private List<Category> categories;
12 }
```

Изворен код 5.19: Пример за изворен ентитет на еднонасочна релација повеќе-кон-повеќе

За дефинирање на двонасочна релација повеќе-кон-повеќе потребно е да се дефинира релација и во другата насока, која исто така треба да биде од типот повеќе-кон-повеќе. Во тој случај, означувањето на инверзната страна се постигнува со помош на параметарот `mappedBy` во рамките на анотацијата за релација, па преостанатата страна ќе биде означена како сопственик. Но, останува прашањето кој од двета ентитети да се означи како инверзен? Таа одлука ја носи програмерот и може да се направи по случаен избор.

Според тоа, за да се дефинира двонасочна релација повеќе-кон-повеќе, потребно е да се направат следните чекори:

- Се додава анотацијата `@ManyToMany` кај ентитетот сопственик пред декла-

рацијата на својството од типот колекција од инверзни ентитети. Се препорачува користење на анотацијата `@JoinTable` за да се означи името на сврзувачката табела и нејзините колони.

- Се додава анотацијата `@ManyToMany` во инверзниот ентитет пред декларацијата на својството од типот колекција од сопственик ентитети. Во анотацијата се пренесува и аргументот `mappedBy`, чија вредност е името на својството преку кое се креира релацијата во ентитетот сопственик.

Ако се водиме според оваа дефиниција за двонасочна релација повеќе-кон-повеќе помеѓу класите `Product` и `Category`, одлучувајќи дека `Product` ќе биде ентитет сопственик, тогаш првото правило ќе се примени кај овој ентитет и неговата дефиниција ќе биде како дефиницијата прикажана во кодниот сегмент 5.19. Примената на второто правило кај инверзниот ентитетот `Category`, ќе резултира со кодот даден во изворниот код 5.12.

```

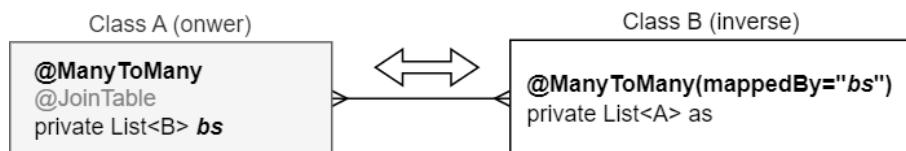
1  @Entity
2  public class Category {
3      @Id @GeneratedValue
4      private int id;
5      private String name;
6      private String description;
7      @ManyToMany(mappedBy="categories")
8      private List<Product> products
9  }

```

Изворен код 5.20: Пример за дестинациски ентитет на двонасочна релација еден-кон-повеќе

Табелите добиени од пресликувањето на ентитетите `Product` и `Category` со двонасочна повеќе-кон-повеќе нема да се разликуваат од табелите добиени со еднонасочна повеќе-кон-повеќе релација.

Правилата за дефиниција на двонасочна релација повеќе-кон-повеќе се шематски прикажани на слика 5-12.

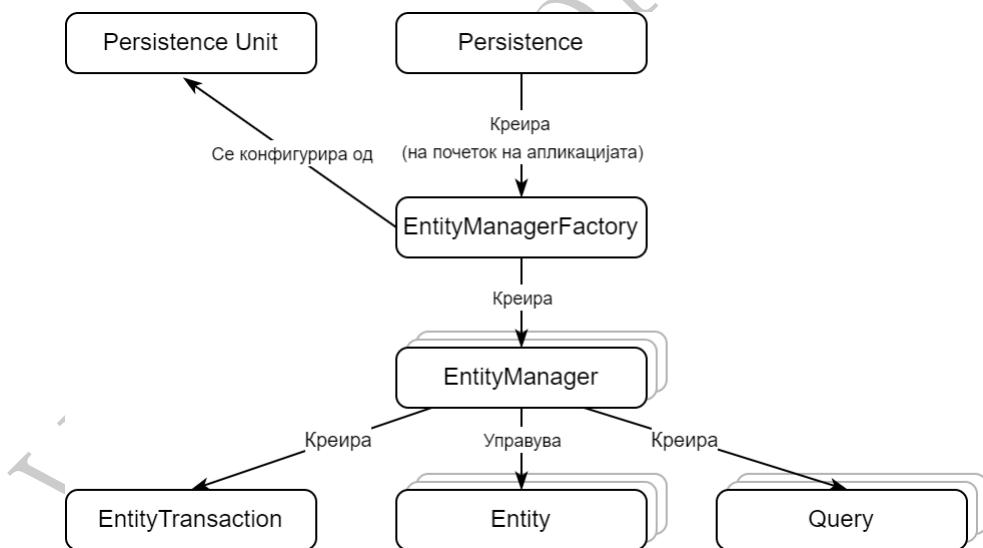


Слика 5-12: Шематски приказ на дефиниција за двонасочна релација повеќе-кон-повеќе

Доколку не е потребна двонасочна релација, тогаш се користи само анотацијата @ManyToMany без аргумент кај сопственикот на релацијата.

5.4 Архитектура на JPA

Целата архитектура на JPA е прикажана на слика 5-13. Според конфигурацијата на перзистентната единица (`jakarta.persistence.PersistenceUnit`) за спецификите на базата, `jakarta.persistence.Persistence` класата креира инстанца од `jakarta.persistence.EntityManagerFactory` при стартувањето на апликацијата. `EntityManager` одржува склад од JDBC конекции кон конфигурираната база кои ги доделува на секој `jakarta.persistence.EntityManager` кој ќе го креира. `EntityManagerFactory` е одговорна за креирање и управување со животниот циклус на `EntityManager`. Секој `EntityManager` управува со ентитети кои се дел од перзистентниот контекст и е одговорен за нивна синхронизација со табелите во базата на податоци. Операциите кон базата ги спроведува преку трансакциите на објект `jakarta.persistence.EntityTransaction` кој овозможува нивно автоматично извршување. Дополнително, преку објектот `jakarta.persistence.Query`, `EntityManager` може да извршува кориснички-дефинирани пребарувања.



Слика 5-13: Составни компоненти на архитектурата на JPA

За да се извршиме интеракција со база на податоци, најпрво треба да се конфигурираат податоците специфични за поврзување со самата база на податоци, како што се име на база, локација, корисничко име, лозинка, итн., како и параметрите специфични за JPA имплементацијата. Овие податоци се дефинираат во форма на посебна **перзистентна единица** (Persistence Unit, англ.), која покрај

параметрите за поврзување со базата на податоци ги содржи и конфигурациите за класите кои ќе бидат третирани како ентитети од страна на `EntityManager`. Во рамките на Spring Boot апликациите, најголемиот дел од овие параметри се однапред конфигурирани со автоматска конфигурација на `spring-data-jpa` модулот. Развивачот треба само да ги дополнi специфичните параметри за поврзување со базата на податоци преку `application.properties` датотеката, како што е прикажано во кодот 5.21. Кaj Spring Boot апликациите не треба експлицитно да ги конфигурираме класите за ентитетите во посебна датотека, бидејќи предефинирано како ентитети ќе бидат регистрирани сите класи од пакетот на апликацијата кои се анотирани со `@Entity`. Доколку сакаме да вклучиме некоја друга патека, тоа може да го сториме со `@EnableJpaRepositories(basePackages = "path.to.the.entity.package")` анотацијата.³

```

1 # datasource config
2 spring.datasource.url=jdbc:postgresql://localhost:5432/e-shop
3 spring.datasource.username=admin
4 spring.datasource.password=ch@ngeMe
5
6 # JPA implementation config
7 spring.jpa.hibernate.ddl-auto=validate
8 spring.jpa.show-sql=true
9 spring.jpa.properties.hibernate.format_sql=true
10 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

```

Изворен код 5.21: Конфигурација за поврзување со PostgreSQL базата на податоци и дел од специфичните параметри за Hibernate имплементацијата

При стартивање на апликацијата, `Persistence` класата креира `EntityManagerFactory` за секоја перзистентна единица. `EntityManagerFactory` креира склад од повеќе JDBC конекции кон базата кои кои му се на располагање да ги доделува на инстанците на `EntityManager`. Затоа, во апликацијата можат истовремено да постојат повеќе инстанци на `EntityManager` кои истовремено ќе управуваат со ентитетите кои се синхронизираат со една конфигурирана база на податоци.

Со дефиницијата на JPA ентитети се овозможува пресликување помеѓу класите и нивните инстанци во релационите бази во форма на табели и редици од

³Доколку имаме потреба да се поврземе со повеќе бази на податоци од една веб апликација, потребно е да се конфигурираат повеќе перзистентни единици. Оваа конфигурација се реализира преку креирање на соодветните Spring бинови за JPA објектите и дефинирање на информации за поврзување со различните бази. Сепак, оваа конфигурација е надвор од доменот на оваа книга.

табелите, соодветно. Ентитетите претставуваат управувани објекти, а за управување со нивниот животниот циклус и синхронизација со базата на податоци се користат имплементациите на интерфејсот `jakarta.persistance.EntityManager`. `EntityManager` обезбедува методи за креирање, зачувување, бришење и читање на ентитети кон и од базата на податоци. Имплементациите на `EntityManager` се добиваат од библиотеките кои ја имплементираат JPA спецификацијата (пр. Hibernate). Во продолжение, кога ќе зборуваме за `EntityManager`, ќе го дефинираме однесувањето кое треба да е имплементирано од овие библиотеки.

Сите ентитети со кои управува `EntityManager` се чуваат во посебна мемориска јединица т.н. **перзистентен контекст** (анг. persistence context). Перзистентниот контекст преставува еден вид на кеш меморија во која се чуваат ентитети кои треба да се синхронизираат со базата. `EntityManager` ги подржува следните операции со перзистентниот контекст:

- Додавање на нови ентитети во перзистентниот контекст преку методот `EntityManager.persist()`.
- Измена на постоечки ентитети во перзистентниот контекст со методот `EntityManager.merge()`.
- Бришење на ентитети од перзистентниот контекст со методот `EntityManager.remove()`.
- Вчитување на ентитети во перзистентниот контекст со методот `EntityManager.find()`.
- Извршување на уникатни пребарувања во перзистентниот контекст со методот `EntityManager.createQuery()`.
- Зачувување на сите промени направени во ентитетите во перзистентниот контекст во базата на податоци со методот `EntityManager.flush()`.
- Бришење на сите ентитети од перзистентниот контекст со методот `EntityManager.clear()`.
- Промена на модот на работа со перзистентниот контекст во `read-only` со методот `EntityManager.lock()`.

Овие операции од `EntityManager` овозможуваат управување со ентитетите во перзистентниот контекст и нивно синхронизирање со базата на податоци.

`EntityManager` ги извршува операциите кон базата користејќи трансакциски објект `jakarta.persistance.EntityTransaction`. Со ова се постигнува атомично извршување на повеќе операции, со што се гарантира конзистентноста на податоците. Доколку барем една од операциите е неуспешна, тогаш ниту една од операциите нема да се примени кај базата на податоци. Секој `EntityManager` креира само еден објект `EntityTransaction` преку кој ги извршува сите потребни операции со ентитетите.

JPA (Jakarta Persistence API) Query е механизам во JPA кој овозможува извршување на прашања (анг. queries) над објектно-релационата мапирана база на податоци. Тоа значи дека JPA Query овозможува извршување на различни типови прашања, како што се SELECT, UPDATE, DELETE, и други, врз податоците кои се складирани во базата на податоци. Овие објекти се креираат од страна на EntityManager.

5.5 Користење на EntityManager

Сосема природно се поставува прашањето до кој од сите овие објекти треба да се добие пристап за да се работи со ентитети и на кој начин да се добие неговата референца?

Одговорот на ова прашање зависи од тоа на кој начин е управуван EntityManager. Постојат два начини на управување:

- **EntityManager урпавуван од апликација** (анг. application managed). За да работи со EntityManager, апликацијата, односно самиот развивач на софтвер мора (1) да добие референца од EntityManagerFactory (со инјектирање преку анотацијата @PersistenceUnit, како во примерот подолу); (2) со помош на инстанцата да креира EntityManager и (3) да управува со животен циклус на креираниот EntityManager. Тоа значи дека (4) треба да креира трансакциски објект EntityTransaction, (5) да ја иницијализира трансакцијата за работа (преку EntityTransaction.begin()), (6) да ги повика методите од EntityManager за работа со ентитетите, (7) потоа да поднесе барање сите промени да се синхронизираат со базата преку трансакцискиот објект (EntityTransaction.commit()) и на крајот (8) да го затвори самиот EntityManager (EntityTransaction.close()). Иако дава поголема контрола, ваквиот начин е многу покомплексен, отвора простор за грешки при кодирањето и ја намалува ефикасноста на програмерот.
- **EntityManager управуван од контејнер** (анг. container managed). Во овој случај се користи надворешен контејнер, кој подржува креирање на инстанца од EntityManager. Контејнерот управува со животниот циклус на креираната инстанца, па сè што треба да направи програмерот е да добие референца од EntityManager и да ги изврши потребните промени врз ентитетите кои потоа ќе се синхронизираат со базата. За сите останати операции се грижи самиот контејнер.

Бидејќи главниот фокус на оваа книга се веб апликаци во работната рамка Spring, кои по природа се управувани од Spring контејнер, ќе се користи EntityManager управуван од контејнерот. Во тој случај, инстанцата од

`EntityManager` се добива со помош на инјектирање користејќи ја анотацијата `@PersistanceContext` на следниот начин:

```

1  @PersistanceUnit
2  EntityManagerFactory emf;
3
4  @PersistanceContext
5  EntityManager em;
```

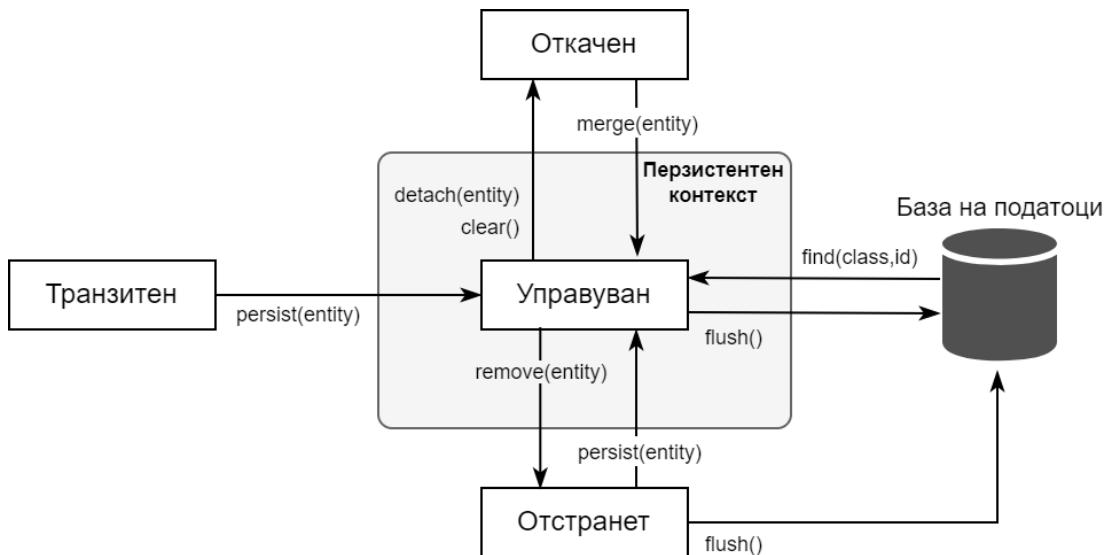
Доколку даден метод од Spring Boot апликација има потреба да работи со ентитети, тогаш тој метод се анотира со `@Transactional`, што му кажува на Spring контејнерот дека пред повикот на методот треба да креира трансакциски објект и да се иницијализира трансакција. Откако ќе заврши методот, контејнерот ги испраќа сите барања до базата преку една атомична трансакција. Пред секое извршување на метод означен со `@Transactional`, контејнерот проверува дали веќе постои отворена трансакција од друг таков метод. Ако не постои, креира нова, а ако постои ја користи постоечката. Со тоа, секогаш кога методот ќе почне да се извршува, веќе постои трансакциски објект кој е отворен и подготвен да проследи барања до базата. Анотацијата `@Transactional` и операциите кои се извршуваат пред да се влезе во методот и откако ќе се излезе од него се типишен пример за користење на концептот на аспектно ориентирано програмирање (Aspect Oriented Programming AOP) на Spring.

Во рамките на `@Transactional` методите се повикуваат акции за интеракција со базата на податоци преку повици на методи од `EntityMangaer` за различни ентитети, но тие не се синхронизираат со базата во моментот на повикот, туку дури откако методот ќе ја изврши последната линија код. Во тој момент, контејнерот ги поднесува барањата за извршување до базата (со повик на `EntityTransaction.commit()` кај трансакцисиот објект), по што го затвора трансакцискиот објект и `EntityManager` и го ослободува перзистентниот контекст. Во случај кога методот означен со `@Transactional` повикуваа друг метод исто така означен со `@Transactional`, кај вториот метод се користи истиот трансакциски објект за работа со ентитети и ентитетите се синхронизираат со базата преку една иста трансакција откако првиот повикан метод ќе заврши. Кај веб апликациите каде барањата се обработуваат од сервлети (и контролерите во Spring се повикани од `DispatcherServlet`), за секое барање се креира нова инстанца на `EntityManager`, а откако одговорот ќе биде генериран, таа инстанца се затвора.⁴.

⁴Ако не сакаме да се отвора трансакција за секое барање, потребно е во `application.properties` да конфигурираме `spring.jpa.openInView=false`

5.6 Животен циклус на ентитет

Секој ентитет може да се наоѓа во неколку различни состојби во зависност од операцијата преку која е добиен (читање од база, инстанцирање или извршување на некој од методите од EntityManager). Животниот циклус на ентитетот е дефиниран со дијаграмот на состојби низ кои може да се поминува ентитетот во рамките на една апликација и е прикажан на слика 5-14.



Слика 5-14: Модел на животен циклус на ентитет

Според сликата, ентитетот може да биде во една од следните состојби:

- **Транзитен** (анг. *Transient*) - ентитетот е надвор од перзистентиот контекст бидејќи е добиен со директно инстанцирање преку конструктор од класата `new EntityClass()`.
- **Управуван** (анг. *Managed*) - ентитетот е во перзистентиот контекст и управуван од страна на EntityManager. Било која промена врз ентитетот ќе се синхронизира со базата на податоци.
- **Откачен** (анг. *Detached*) - ентитетот бил во перзистентиот контекст, но е отстранет преку некој од меетодите на EntityManager. Промените врз ентитетот во оваа состојба нема да се синхронизираат со базата сè додека не се врати назад во перзистентниот контекст (ако воопшто се врати во контекстот).
- **Отстранет** (анг. *Removed*) - ентитетот е отстранет од контекстот, но е означен за бришење. Ако не се врати назад во перзистентиот контекст при синхронизацијата со базата ќе биде избришан од соодветната редица во табелата.

За премин од една состојба во друга се користат методите на интерфејсот `EntityManager`.

5.6.1 Креирање на управувани ентитети

Секој ентитет кој се креира преку конструкторот на класата со која е дефиниран се наоѓа во *транзитна* состојба и не е поврзан со перзистентниот контекст, а со тоа не е синхронизиран со базата на податоци. За да стане *управуван* ентитет, потребно е да се внесе во перзистентниот контекст преку методот `em.persist(entity)` каде `entity` е *транзитна* или *остранета* инстанца на ентитетот.

Во изворниот код 5.22 е даден пример за метод кој креира ентитет од класата `Customer`. Методот ја користи анотацијата `@Transactional` за Spring контејнерот да подготви трансакција пред да почне да се извршува методот и да ја изврши и затвори откако методот ќе изелзе. Откако ќе се креира инстанцата со празен конструктор, се поставуваат својствата, а потоа таа се внесува во перзистентниот контекст преку методот `em.persist()`, како што е прикажано во изворниот код 5.22.

```

1  @PersistanceContext
2  EntityManager em;
3
4  @Transactional
5  public Customer create(String firstName, String lastName){
6      Customer c = new Customer();
7      c.setFirstName(firstName);
8      c.setLastName(lastName);
9      return em.persist(c)
10 }
```

Изворен код 5.22: Пример за креирање на управуван ентитет.

Важно е да се забележи дека откако ќе се повика методот `em.persist()`, промената не се применува инстантно во базата на податоци. Иако контејнерот обезбедил инстанца на `EntityManager` и креирал транскациски објект преку кој ќе се пренесат барањата во базата, методот за поднесување на тие барања во формат на SQL кон базата (`EntityTransaction.commit()`) ќе биде повикан од контејнерот дури откако методот ќе заврши.

5.6.2 Вчитување и освежување на ентитет од базата на податоци

За да се чита постоечки ентитет од базата на податоци според идентификатор се користи методот `em.find(entityClass, id)` каде `entityClass` е класата на која припаѓа ентитетот, а `id` е вредноста на неговиот единствен идентификатор. За да се направи дистрибуција за кој ентитет станува збор, односно во која табела од базата да се пребарува редицата со дадениот идентификатор `id`, се користи класата на пребаруваниот ентитет. Дополнително, класата на ентитетот помага на методот да изврши конверзија на податоците добиени како одговор од базата во ентитет од таа класа. Ако во табелата се пронајде редица со бараната вредност на идентификаторот, тогаш методот враќа инстанца на ентитет кој го сместува во перзистентниот контекст. Во спротивно, методот ќе фрли исклучок.

Пример за пребарување на ентитетот `Customer` преку `EntityManager` е даден во изворниот код [5.23](#).

```

1  @Transactional
2  public Customer findById(int id){
3      return em.find(Customer.class,id)
4 }
```

Изворен код 5.23: Пример за читање на ентитет од база.

Покрај методот за пребарување на ентитет во базата според идентификатор, постои и метод за пребарување на ентитет во перзистентниот контекст: `em.contains(entityInstance)`, кој враќа `true` секогаш кога `entityInstance` е во управувана состојба, односно е внесен во перзистентниот контекст. Ако ентитетот е *откачен* или пак ако станува збор за нов *транзитен* ентитет кој сеуште не е внесен во перзистентниот контекст, тогаш методот ќе врати вредност `false`.

Откако еден ентитет ќе се смести во перзистентниот контекст, неговите вредности се синхронизираат со вредностите во базата на податоци. Промените направени директно во базата на податоци не се синхронизираат со постоечките ентитети во перзистентниот контекст. За да се изврши синхронизирање на новите вредности од база со соодветните ентитети во перзистентниот контекст се користи методот `em.refresh(entityInstance)`, каде `entityInstance` е управуваниот ентитет за кој сакаме да ја освежиме вредноста според состојбата во базата на податоци. При тоа, било кои промени кои биле направени врз ентитетите во перзистентниот контекст ќе бидат препокриени од вредностите добиени од базата. Овој метод е корисен во ситуациите кога управуваниите ентитети се наоѓаат подолго време во перзистентниот контекст, па постои можност нивнате вредности да не соодвет-

суваат со пресликаните вредност во базата поради голема конкурентност.

5.6.3 Ажурирање на ентитет

Управуван ентитет може едноставно да се менува и синхронизира со базата со самото менување на неговите својства. Пример за промена на ентитетот `Customer` е даден во изворниот код 5.24. Со самиот факт што ентитетот е добиен преку методот `find()`, тој станува управуван, па затоа било која промена на неговите својства се синхронизира со базата на податоци при затворањето на трансакцијата.

```

1 @Transactional
2 public Customer update(int id, String firstName, String lastName){
3     Customer c = em.find(Customer.class,id);
4     if (c != null){
5         c.setFirstName(firstName);
6         c.setLastName(lastName);
7     }
8     return c;
9 }
```

Изворен код 5.24: Пример за ажурирање на ентитет во база.

Кaj веб апликациите, постојат сценарија во кои се повикува контролер на кој му пренесува инстанца од ентитет која треба да се зачува во база и кој содржи идентификатор. На пример, корисникот го променил своето презиме во прелистувачот, а потоа објектот за тој `Customer` ентитет во json формат го испраќа до сервисот на веб апликацијата за ажурирање во базата на податоци. Пристигнатиот објект не е управуван ентитет, но бидејќи бил испратен до клеинтот (некогаш бил во управувана состојба) потребно е да се врати повторно во перзистентниот контекст за да се синхронизира со базата. Тоа се постигнува со помош на методот `em.merge(entityInstance)`, кој го внесува откачен ентитет `entityInstance` во перзистентниот контекст, а со тоа неговите промени се синхронизираат со базата.

Во изворниот код 5.25 е даден пример на ажурирање на откачен ентитет од класата `Customer` преку негово враќање во управувана состојба.

```

1 @Transactional
2 public void update(Customer c){
3     if (c.id != null){
4         em.merge(c)
5     }
}
```

6 }

Изворен код 5.25: Пример за ажурирање на откачен ентитет во база.

5.6.4 Бришење на ентитет

За да се отстрани ентитет од базата на податоци, тој треба да биде најпрво *управуван*, а потоа да се отстрани од перзистентниот контекст со користење на методот `em.remove(entityInstance)`. На овој начин, при синхронизацијата со базата ќе биде отстранета редицата во која е пресликан тој ентитет. Важно е да се направи дистинкција помеѓу *отстранет* (анг. removed) и *откачен* (анг. detached) ентитет. И двата типови на ентитети се извадени надвор од перзистентниот контекст, но *отстранетиот* ентитет е обележан за бришење од база, а *откаченниот* е едноставно исклучен од синхронизација со базата и било какви промени врз него не влијаат на базата. Еден *управуван* ентитет се откачува од перзистентниот контекст со помош на методот `em.detach(entity)`.

Пример за бришење на ентитетот `Customer` е дадена во изворниот код 5.26. Откако ќе се добие инстанцата `c` на ентитетот преку методот за пребарување според идентификаторот, се повикува методот `em.remove(c)` кој ќе го обележи објектот за бришење. Бришењето, како и сите претходни операции за модификација на ентитатите, не се извршува иstantно, туку откако ќе се затвори трансакцијата.

```

1 @Transactional
2 public void delete(int id){
3     Customer c = em.find(Customer.class,id);
4     if (c != null){
5         em.remove(c)
6     }
7 }
```

Изворен код 5.26: Пример за бришење на управуван ентитет од база.

Доколу еден ентитет `entityInstance` се острани од контекстот за подоцна да биде избришан од базата преку `em.remove(entityInstance)`, тој може повторно да се врати во перзистентниот контекст ако се повика методот `em.persist(entityInstance)`. Во ваков случај ентитетот нема да се избрише.

5.6.5 Методи за работа со перзистентен контекст

Од претходните примери може да се види дека `EntityManager` нуди методи за менување на состојбата на еден ентитет во однос на неговата припадност во

перзистентниот контекст, но без разлика за каква промена станува збор, кај `EntityManager` управуван од контејнерот промената во база не се реализира веднаш по повикувањето на некој од неговите методи, туку дури откако методот во кој се повикуваат тие методи ќе врати резултат. Но, што доколку е потребно ентитетите да се перзистираат веднаш, пред тоа да го направи контејерот по излезот од методот? Во тој случај, се користи методот за присилна синхронизација `em.flush()`. Овој метод се повикува без аргументи и прави моментална синхронизација на сите ентитети во перзистентниот контекст.

После секое извршување на барања за читање или промени во базата, контејерот го затвора `EntityManager` и ги ослободува сите перзистентни ентитети од перзистентниот контекст. Оваа операција ја постигнува со методот `em.clear()` кој има слична функција како и методот `em.detach(entityInstance)`, но за разлика од него, наместо да отстрани специфичен ентитет од перзистентниот контекст, тој ги отстранува сите ентитети кои се наоѓаат во контекстот. Исфрелените ентитети остануваат во меморијата, но се надвор од перзистентниот контекст во *откачена* состојба. Употребата на овој метод не е поврзана со контејнерот и може да биде повикан секогаш кога е потребно присилно празнање на перзистентниот контекст.

5.6.6 Извршување на кориснички дефинирани прашања

Освен методите за извршување на операции за вчитување, креирање, менување и бришење на ентитети, JPA овозможува и извршување на кориснички дефинирани барања напишани во JPQL (Java Persistent Query Language) или во изворен SQL преку класи изведени од класата `jakarta.persistence.Query`. Станува збор за класите:

- `TypedQuery` - се користи за извршување на динамички прабаруава дефинирани со JPQL
- `NamedQuery` - се користи за извршување на статички прабарувања дефинирани со претходно зачувани и именувани JPQL
- `NativeQuery` - се користи за извршување на динамички прабаруава дефинирани со SQL

Интерферијсот `EntityManager` нуди методи за креирање соодветна `Query` класа за секој од наведените типови:

- `TypedQuery q = em.createQuery(query, resultClass)` - се користи за креирање на динамичко пребараување од типот `TypedQuery` каде `query` е JPQL прашањето за пребараување, а `resultClass` е класата во која треба да се конвертираат податоките кои ќе ги врати пребараувањето (на пример класата на ентитотот или некоја од обвиканите класи за податочни типови)

- `NamedQuery q=em.createNamedQuery(namedQuery, resultClass)` - се користи за креирање на `NamedQuery` за претходно дефинирано именувано прашање
- `NativeQuery q = em.createNativeQuery(query, resultClass)` - се користи за креирање на динамичко пребарување од типот `NativeQuery` каде `query` е SQL пребарување

Секоја од овие класи го имплементира `jakarta.persistence.Query` интерфејсот и нуди методи за извршување на пребарувањето кои се разликуваат во зависнот од типот на податоци и бројот на елементи кои треба да ги врати пребарувањето:

- `q.getResultList()` - го извршува барањето и враќа листа од ставки (ентитет или друг тип на податоци)
- `q.getSingleResult()` - го извршува барањето, но враќа само еден елемент
- `q.executeUpdate()` - извршува барање за ажурирање или бришење кое не враќа конкретни резултати

Во изворниот код 5.27 е даден пример за извршување на кориснички дефинирано барање користејќи го јазикот JPQL. Во кодот, најпрво се креира барање со параметар кое ќе ги врати сите ентитети од типот `Customer` кои имаат вредност на својството `firstName` како пренесениот параметар. Потоа, се поставува вредноста на параметарот и се извршува барањето.

```

1 public List<Customer> findByFirstName(String firstName){
2     String query = "select a from Customer a where a.firstName=:firstName "
3     return em.createQuery(query,
4         Customer.class).setParameter("firstName",firstName).getResultList();
5 }
```

Изворен код 5.27: Пример за извршување на кориснички дефинирано барање со JPQL

5.6.7 Вежба: креирање и користење на репозиториум за работа со ентитети

За да разгледаме пример со примената на методите од `EntityManager` за работа со ентитети, ќе креираме посебен репозиториум кој ќе ги содржи претходно описаните методи за CRUD операции, но и дополнителни операции за пребараување на ентитетот `Address`. Користејќи го вметнувањето на зависности кај Spring, овој репозиториум ќе го вметнеме во апликациската класа на проектот и преку команда линија ќе ги повикаме неговите методи за да креираме конкретни ентитети и да извршиме различни операции врз нив.

TODO: За таа цел отворете го проектот во кој веќе се дефинирани ентитетите, задржувајќи ги истите поставки на JPA првојадер. Креирајте пакет со име *repository* во кој ќе креирате класа со име **AddressRepository** со сдржина дадена во изворниот код [5.28](#).

```
1 @Repository
2 public class AddressRepository {
3     @PersistenceContext
4     EntityManager em;
5
6     @Transactional
7     public Address findById(int id){
8         return em.find(Address.class,id);
9     }
10    @Transactional
11    public Address create(Address address){
12        em.persist(address);
13        return address;
14    }
15    @Transactional
16    public Address update(Address address){
17        em.merge(address);
18        return address;
19    }
20    @Transactional
21    public void delete(int id){
22        Address address = em.find(Address.class,id);
23        em.remove(address);
24    }
25    @Transactional
26    public List<Address> findAll(){
27        TypedQuery<Address> query = em.createQuery("select a from Address
28            a", Address.class);
29        return query.getResultList();
30    }
31    @Transactional
32    public List<Address> findByPostalCode(int pc){
33        TypedQuery<Address> query = em.createQuery("select a from Address a
34            where a.postalCode = :postalCode", Address.class);
35        query.setParameter("postalCode", pc);
36        return query.getResultList();
```

```
35     }
36 }
```

Изворен код 5.28: Содржина на репозиториумот AddressRepository за работа со ентитетот Address

Класата AddressRepository е означена како репозиториум со помош на анотацијата @Repository. Оваа анотација е изведена од @Component и се користи за регистрирање на бинови во Spring апликацискиот контекст кои се користат во податочниот слој на апликациите, одговорен за работа за податоци. Се разликува од изведената анотација @Component по тоа што вклучува исклучоци поврзани со перзистентност и ги проследува како унифицирани исклучоци во Spring.

Со оглед на тоа што станува збор за EntityManager контролиран од Spring контејнерот, ја користиме анотацијата @PersistenceContext за да вметнеме инстанца em од EntityManager (kreirana од страна на Spring преку EntityManagerFactory). Пред секој од методите во репозиториумот се користи и анотацијата @Transactional за да се нагласи дека пред да се повика методот потребно е да се провери дали веќе има креиран и иницијализиран трансакциски објект кој може да се искористи за атомично извршување на сите операции (ако не постои, се креира, а ако постои се користи постоечкиот). Откако ќе заврши методот, доколку тој ја иницијализирал трансакцијата ги извршува сите операции, ја затвора конекцијата и ја ослободува инстанцата на em.

Во изворниот код 5.35 е дадена содржината на апликациската класа на проектот во која најпрво се декларира променлива од типот AddressRepository, а потоа се дефинира конструктор преку кој се прави вметнување на оваа зависност, односно Spring контејнерот доделува вредност која претставува инстанца од класата AddressRepository. Во рамките на класата се дефинира и методот runAddressExample() кој ќе се повика за да се изврши примерот за работа со инстанци на ентитетот Address користејќи го рапозиториумот AddressRepository.

Бинот CommandLineRunner исто така се вметнува во апликацискиот контекст и истиот го користиме за во него да го повикаме методот runAddressExample().

```
1 @SpringBootApplication
2 @EntityScan(basePackages = {"mk.ukim.finki.wp.ch4examples"})
3 public class Ch4ExamplesApplication {
4     final AddressRepository addressRepository;
5     public Ch4ExamplesApplication(AddressRepository addressRepository) {
6         this.addressRepository = addressRepository;
7     }
8     public static void main(String[] args) {
9         SpringApplication.run(Ch4ExamplesApplication.class, args);
```

```
10    }
11    @Bean
12    public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
13        return args -> {
14            runAddressExample()
15        };
16    }
17    public void runAddressExample(){
18        Address address;
19        address = new Address();
20        address.setStreet("Happiness Blvd.");
21        address.setCity("Joy town");
22        address.setPostalCode(1000);
23        addressRepository.create(address);
24        System.out.println("----New address----");
25        System.out.println(address.toString());
26
27        address.setCountry("Dreamland");
28        addressRepository.update(address);
29        System.out.println("----Updated address----");
30        System.out.println(address.toString());
31
32        address = new Address();
33        address.setStreet("Fun Av.");
34        address.setCity("Joy town");
35        address.setPostalCode(1000);
36        addressRepository.create(address);
37
38        address = new Address();
39        address.setStreet("Forest St.");
40        address.setCity("Future city");
41        address.setPostalCode(9000);
42        addressRepository.create(address);
43
44        System.out.println("----All addresses----");
45        System.out.println(addressRepository.findAll());
46
47        System.out.println("----Filtered addresses----");
48        System.out.println(addressRepository.findByPostalCode(1000));
49
50    }
```

51 }

Изворен код 5.29: Содржина на апликациска класа за извршување на команди од репозиториумот `AddressRepository`

Во методот `runAddressExample` од изворниот код 5.35 најпрво се креира инстанца од ентитетот `Address` чии својства се исполнуваат со вредности преку методите за поставување својства на класата. Оваа инстанца не е перзистентна, сè додека не се повика методот `create()`. Ако се изврши апликацијата, тогаш на командната линија ќе биде испишана следната содржина:

```
1 ----New address-----
2 Address(id=1, street=Happiness Blvd., city=Joy town, postalCode=1000,
country=null)
```

Може да се забележи дека иако ентитетот нема вредност за идентификатор пред да биде перзистиран, но потоа `EntityManager` се погрижил да креира уникатен идентификатор, да го додели на својството `id` и да го перзистира објектот во база. Сите овие операции можат да се идентификуваат на конзолата како конкретни SQL наредби кои промајдерот ги испраќа до базата:

```
1 Hibernate: select nextval ('hibernate_sequence')
2 Hibernate: insert into address (city, country, postal_code, street, id)
values (?, ?, ?, ?, ?)
```

Според излезот, Hibernate ја чита следната вредност на секвенцата `nextval` која ќе ја искористи како вредност на `address.id`, а потоа ја внесува инстанцата на ентитетот `address` во табелата `address` користејќи ја SQL наредбата `insert`. Откако ќе заврши методот `create()` на репозиториумот, завршува и трансакцијата, се брише перзистентиот контекст и се затвора инстанцата на `EntityManager`, па според тоа инстанцата `address`, иако се наоѓа во меморијата, не е повеќе перзистентна. Ако се направи нова промена на вредноста на некое нејзино својство, таа нема да се преслика во базата на податоци. Тоа може да се заклучи ако во линијата 24, се постави breakpoint и апликацијата се изврши така што ќе биде овозможено дебагирање (debug). Ако се погледне содржината на табелата `address` во моментот кога ќе застане извршувањето во линијата со прекин, може да се забележи дека колоната `country` има вредност `null` иако во линијата 23 на својството на ентитетот му се доделува вредност. Промената ќе се синхронизира дури откако ќе се повика методот `addressRepository.update()` според имплементацијата која ја диктира анотацијата `@Transactional` која се користи во дефиницијата на самиот метод. На конзолата ќе се испише ажурираниот ентитет, односно пораката:

```

1 ----Updated address-----
2 Address(id=1, street=Happiness Blvd., city=Joy town, postalCode=1000,
          country=Dreamland)

```

Ако се погледнат SQL командите кои ги повикал Hibernate

```

1 Hibernate: select ... from address address0_ where address0_.id=?
2 Hibernate: update address set city=?, country=?, postal_code=?, street=?
          where id=?

```

може да се види дека преку методот `merge(entity)` Hibernate најпрво го пребарува ентитетот според *id* со помош на SQL *select* за да го вметне во перзистентниот контекст и да ја преземе неговата последна состојба зачувана во база. Потоа ја ажурира неговата состојба со новите вредности на *entity* (ова ажурирање се прави во меморијата, па затоа нема код во конзолата) и на крајот ја синхронизира новата состојба на ентитетот со базата на податоци преку SQL *update*.

Во продолжение се креираат уште две инстанци (линија 28-38) кои Hibernate ги перзистира во базата со помош на следните SQL наредби:

```

1 Hibernate: select nextval ('hibernate_sequence')
2 Hibernate: insert into address (city, country, postal_code, street, id)
          values (?, ?, ?, ?, ?)
3 Hibernate: select nextval ('hibernate_sequence')
4 Hibernate: insert into address (city, country, postal_code, street, id)
          values (?, ?, ?, ?, ?)

```

За приказ на сите инстанци на ентитетот `Address` го повикуваме методот `addressRepository.findAll()` (линија 41) за чија имплементација искористивме објект `TypedQuery` со зададена JPQL наредба за пребарување. Резултатот ќе биде листа од сите внесени објекти:

```

1 ----All addresses-----
2 [Address(id=1, street=Happiness Blvd., city=Joy town, postalCode=1000,
          country=Dreamland),
3  Address(id=2, street=Fun Av., city=Joy town, postalCode=1000,
          country=null),
4  Address(id=3, street=Forest St., city=Future city, postalCode=9000,
          country=null)]

```

Оваа листа се добива на тој начин што Hibernate ја преведува зададената JPQL наредба `"select a from Address a"` во следната SQL наредба:

```

1 Hibernate: select address0_.id as id1_0_, address0_.city as city2_0_,
          address0_.country as country3_0_, address0_.postal_code as postal_c4_0_,

```

```
address0_.street as street5_0_ from address address0_
```

На крајот (линија 43) го повикуваме методот `addressRepository.findByPostalCode(postalCode)` за пребарување ентитети кои имаат вредност на својството `postalCode` како и вредноста која се пренесува како аргумент на методот. И за оваа имплементација користевме `TypedQuery` со зададена JPQL наредба, но овој пат, наредбата содржи и параметар за пребарување. Во примерот се пребаруваат и печатат сите ентитети кои имаат вредност 1000 на својството `postalCode`, па затоа испечатената листа на адреси ќе биде

```
1 ----Filtered addresses-----
2 [Address(id=1, street=Happiness Blvd., city=Joy town, postalCode=1000,
3           country=Dreamland),
3  Address(id=2, street=Fun Av., city=Joy town, postalCode=1000,
4           country=null)]
```

Резултатот од преводот на JPQL наредба `"select a from Address a where a.postalCode = :postalCode"` од страна Hibernate ќе биде следната SQL наредба:

```
1 Hibernate: select ... from address address0_ where address0_.postal_code=?
```

5.7 Работа со ентитети во релација

5.7.1 Читање на ентитети во релација

Ако постојат релации помеѓу ентитеите, тогаш при работа со нивните инстанци потребно е да се земат во предвид и релациите и нивните насоки. Како што видовме претходно, кај релациите изворниот ентитет „знае“ за постоењето на дестинацискиот ентитет, па затоа инстанцата или листата на инстанци од дестинацискиот ентитет може едноставно да се добие со методите за читање кои се дел од самата дефиниција на класата (getter).

Постојат два начини на однесување при вчитување на дестинациските ентитети од релацијата:

- *LAZY*, односно мрзеливо однесување каде дестинацискиот ентитет или листата од ентитети не се читаат од базата со самото читање на изворниот ентитет. За да се повлечат ентитетите, потребно е експлицитно да се повикава методот за читање на својството, по што следни ново барање до базата преку кое `EntityManager` ги чита. Ова однесување е предфинирано кај кон-повеќе врските (еден-кон-повеќе и повеќе-кон-повеќе).

- *EAGER*, односно нетреливо однесување каде дестинацискиот ентитет или листата од ентитети се вчитуваат од базата во перзистентниот контекст веднаш по вчитувањето на изворниот ентитет. Ова однесување е предфинирано кај кон-еден врските (еден-кон-еден и повеќе-кон-еден)

Предефинираниот начин на однесување за секоја релација може да се промени во самата нејзина анотација преку поставување на аргументот `fetch` на вредност `FetchType.LAZY` или `FetchType.EAGER`. Од аспект на перформанси, `FetchType.LAZY` нуди бенефит бидејќи кога се чита поголема листа на изворни ентитети од базата не се генерираат дополнителни барања за читање на сите нивни листи од дестинациски објекти. Со тоа се намалува обемот на работа на податочниот сервер, количината на податоци кои се разменуваат со апликацискиот сервер како и количината на мемориски простор кој се користи кај апликацискиот сервер за складирање на сите овие објекти.

Да се навратиме на примерот за едносачочна еден-кон-повеќе релација помеѓу ентитетите `Customer` и `Order` дадени во изворните кодови [5.14](#) и [5.15](#). Предефинираното однесување на оваа релација во ентиетот `Customer` е *LAZY*, што би било еквивалентно како при дефиницијата на релацијата да се постави

```

1 @OneToMany(fetch=FetchType.LAZY)
2 private List<Order> orders = new ArrayList<>();

```

Нека `customer` е инстанца од ентиетот `Customer` која во својството `orders` веќе има назначено листа од инстанци од типот `Order`. Доколку се побара оваа инстанца од база преку методот `find(id, Customer.class)` на `EntityManager`, тогаш ќе се креира SQL барање до базата кое ќе ги повлече сите својства на класата, но нема да ги повлече нарачките кои ги направил тој клиент.

За да се добијат нарачките кои ги направил клиентот се повикува методот `customer.getOrders()`. За потсетување, иако експлицитно не го дефинираме овој метод во класата, тој е креиран во позадина поради користењето на анотацијата `@Data` која е дел од пакетот `lombok`. Резултатот од овој повик е извршување на ново SQL барање кое ќе ги излиста сите нарачки поврзани со тој клиент.

Во примерот за двонасочна релација помеѓу ентиетите `Customer` и `Order`, релацијата од `Order` кон `Customer` е повеќе-кон-еден. Предефинираното однесување на релацијата во оваа насока е *EAGER*, односно би било еквивалетно ако при нејзиното дефинирање во изворниот код [5.12](#) се постави

```

1 @ManyToOne(fetch=FetchType.EAGER)
2 @JoinColumn(name = "customer_id")
3 private Customer customer;

```

Ваквото однесување значи дека кога ќе се прочита инстанца `order` на `Order` се генерира SQL барање кое ги повлекува колоните на табелата `eshop_order` од реди-

цата која одговара на таа нарачка, но и сите клони на табелта `customer` кои одговараат на редицата со идентификатор чија вредност е иста како и `customer_id`. Ако однесувањето се промени да биде *LAZY*, тогаш се генерира SQL барање кое повлекува податоци само од табелата `eshop_order`, а за да се добијат податоците кои одговараат на клиентот на таа нарачка, потребно е експлицитно да се повика `order.getCustomer()`. Резултатот од овој повик е SLQ барање кое ќе ја прочита редицата од табелата `customer` со идентификатор чија вредност одговара на `customer_id` од табелата `eshop_order` за таа нарачка.

5.7.2 Уредување на ентитети во релација

Уредувањето на ентитети во релација бара поголемо внимание во однос на читањето бидејќи при промена, додавање и отстранување на ентитетите, потребно е да се дефинира како ќе се однесува релацијата, односно на кој начин методите за промена на состојбата на изворниот ентитет ќе се пренесат на дестинациските ентитети. На пример, ако на еден транзијетен изворен ентитет се додадат нови транзитни инстанци на дестинациски ентитети и ако се повика методот `em.persist()` за негово перзистирање, тогаш се поставува прашањето дали методот ќе се пренесе и на дестинациските ентитети? Ако се пренесе, тогаш ќе се перзистираат сите инстанци во базата. Ако не се пренесе, ќе настане проблем бидејќи на веќе постоечкиот перзистентен изворен ентитет се поврзуваат ентитети во транзијетна состојба, што од аспект на базата значи дека постоечка редица во извornата табела се става во релација со непостоечки редици во дестинациската табела. Друг пример на пренесување на состојба е бришење на изворен ентитет со методот `em.remove()` кој веќе е во релација со повеќе инстанци од дестинациски ентитет. Се поставува прашањето дали ќе се избришат и дестинациските ентитети или ќе останат во базата без да бидат во релација со непостоечки изворен ентитет?

Затоа при уредувањето на ентитети во релација, потребно е да се дефинира однесувањето на релацијата, односно начинот на пренесување на методите повикани кај изворниот ентитет врз дестинациските ентитети од таа релација. Предефинираното однесување на релациите е да не се пренесува ниту еден метод за операција со ентитети, па доколку има несовпаѓање на состојбите на ентитетите во текот на извршување на операцијата поради нивната зависност, можно е `EntityManager` да фрли грешка или да резултира со податоци во базата кои ќе останат меѓусебно неповрзани. За да се избегнат несакани ситуации кои би настапиле со предефинираното однесување, при дефиницијата на релациите помеѓу ентитетите потребно е да се специфицира на кој начин состојбата на изворниот ентитет ќе се пренесува на дестинацискиот ентитет.

Како што веќе дефинираме во секцијата 5.6 за животниот циклус на ентитетите, тие можат да се најдат во различни состојби низ кои може да се транзитира преку методите кои се дел од `EntityManager`. Според тоа, може да се дефинираат различни типови на однесување на релацијата во однос на пренос на состојбата на изворниот ентитет кон дестинацискот ентитет од релацијата. Овој начин на пренесување на состојба или методи од `EntityManager` се нарекува **каскада** (анг. cascade) и во зависност од тоа кој метод на изворниот ентитет сакаме да се пренесе и на дестинацискиот ентитет се дефинираат следните типови на каскади:

- *PERSIST* - повик на методот `em.persist()` кај изворниот ентитет се пренесува и кај дестинациските ентитети
- *MERGE* - повик на методот `em.merge()` се пренесува и кај дестинациските ентитети.
- *DETACH* - повик на методот `em.detach()` се пренесува и кај дестинациските ентитети.
- *REMOVE* - повик на методот `em.remove()` се пренесува и кај дестинациските ентитети.
- *REFRESH* - повик на методот `em.refresh()` се пренесува и кај дестинациските ентитети.
- *ALL* - Било која метод за промена на состојбата на изворниот ентитет се пренесува на дестинациските ентитети.

Дефиницијата на однесувањето на релацијата се дефинира преку аргументот `cascade`, кој како вредност може да прими било кој елемент од енумерацијата `CascadeType` со вредности како и горната листа.

Во изворниот код 5.30 е даден пример за едносасочна еден-кон-повеќе релација помеѓу ентитетите `Customer` и `Order` со дефиниција на однесувањето на релацијата.

```

1  @Entity
2  public class Customer {
3      @Id @GeneratedValue
4      private int id;
5      private String firstName;
6      private String lastName;
7      @OneToMany(cascade = CascadeType.ALL)
8      private List<Order> orders = new ArrayList<>();
9  }

```

Изворен код 5.30: Пример за изворен ентитет на еднонасочна релација еден-кон-повеќе

Според дефиницијата на каскадата од типот CascadeType.ALL, ако на една нова инстанца `customer` од ентитетот `Customer` се додаде новокреирана инстанца `order` од ентитетот `Order`, тогаш при перзистирање на `customer` со повик на методот `em.persist(customer)`, методот ќе се пренесе и на инстанцата `order`, односно `EntityManager` освен што ќе внесе нова редица во табелата `customer`, ќе внесе и нова редица во табелата `order` со надворешен клуч `customer_id` кој ќе ја има истата вредност како идентификаторот `id` на ентитетот `customer`. Со тоа, наместо мануелно да ја перзистираме секоја нова додадена инстанца, тоа ќе го направи `EntityManager` во позадина. Ваквиот пренос на состојба се оденсува на каскада од типот CascadeType.PERSIST која е дел од CascadeType.ALL.

Исто така, ако инстанцата `customer` се назначи за бришење со повик на методот `em.remove(customer)`, тогаш не само `customer`, туку и секоја инстанца од `order` од ентитетот `Order` која е дел од листата `orders` ќе се назначи за бришење преку повик на `em.remove(order)`, па при синхронизација со базата ќе биде отстранета и од табелата во која е пресликана. Ваквиот пренос на состојба се оденсува на каскада од типот CascadeType.REMOVE која е исто така дел од CascadeType.ALL.

Во пракса, потребено е каскадата од типот CascadeType.ALL, поточно типот CascadeType.REMOVE внимателно да се употребува бидејќи може да доведе до не-повратно губење на податоци ако по грешка се избрише изворен ентитет кој е во релација со голем број на дестинациски ентитети. Загубите ќе бидат уште поголеми ако станува збор за двонасочна повеќе-кон-повеќе релација каде и двете страни имаат каскада CascadeType.ALL, бидејќи бришењето ќе се пренесува итеративно од една на друга страна во релацијата сè додека не се избришат сите инстанци кои се на директен или индиректен начин зависни од изврната инстанца. За да се избегне пропагација на сите состојби, може да се изберат само одредени типови на каскада при дефиниција на релацијата кои нема да наштетат на податоците. На пример, ако во класата `Customer` се дефинира релацијата:

```
1 @OneToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})
2 private List<Order> orders = new ArrayList<>();
```

промените кај инстанците на `Customer` ќе се пренесат на инстанците од `Order` со кои се во релација само при повик на методите `em.persist()` и `em.merge()`. Ако се повика било кој друг метод, на пример `em.detach()` или `em.remove()`, `EntityManager` ќе го изврши тој метод кај ентитетот `Customer`, но не и кај инстанците на `Order` со кои е во релација.

Друг важен аспект на релацијата е справувањето со отстранети ентитети. Ако кај инстанцата `customer` од ентитетот `Customer` се отстрани инстанца `order` од листата `orders` преку `customer.getOrders().remove(order)`, тогаш нејзиното от-

странување од врската со `customer` не значи и бришење на соодветната редица во базата на податоци. Напротив, ентитетот `order` останува да егзистира без да има релација со изворниот ентитет `customer`, а како вредност на надворешниот клуч `customer_id` во соодветната редица на табелата `eshop_order` се внесува вредноста `null`. Во таква ситуација, ентитетот `order` не е повеќе референциран од ниту еден изворен ентитет `customer`, па затоа се нарекува **сирак** (анг. orphan). Ова предефинирано однесување на врската може да се промени со поставување на аргументот `orphanRemoval` на вредност `true` што ќе предизвика позадинско отстранување на нереференцираните ентитети од релацијата. Според тоа, ако релацијата еден-кон-повеќе помеѓу `Customer` и `Order` се дефинира како

```
1 @OneToOne(orphanRemoval=true, cascade = CascadeType.ALL)
2 private List<Order> orders = new ArrayList<>();
```

тогаш ако се отстрани `order` од листата на нарачки `orders` и ако се перзистира `customer`, EntityManager ќе генерира наредба за отстранување на соодветната редица од табелата `eshop_order`.

Ако ја разгледаме релација помеѓу ентитетите `Customer` и `Order` како двонасочна, тогаш таа ќе се состои од еднонасочна повеќе-кон-еден релација кај ентитетот `Order` веќе дефинирана во изворниот код [5.12](#) и еднонасочна релација еден-кон-повеќе кај ентитетот `Customer`, чија проширена дефиниција е дадена во изворниот код [5.30](#).

Кога се користи двонасочната релација, треба да се земе предвид дека при додавање или отстранување на инстанца од едната страна на релацијата, потребно е да се ажурира референцата и во другата страна на релацијата. На пример, ако се додава инстанца `order` во листата `orders` на инстанца `customer` од ентитетот `Customer`, потребно е да се ажурира и врската во обратната насока, од `order` кон `customer` така што на својството `order.customer` ќе се додаде референца кон инстанцата на изворниот ентитет `customer` преку неговиот метод за поставување (setter) со што двете страни од релацијата ќе имаат меѓусебна врска и ќе можат да си пристапат до својствата. Оваа операција се извршува преку следните линии код:

```
1 order.setCustomer(customer);
2 customer.getOrders().add(order);
```

Слично, при бришење на `order` од листата `orders` потребно е да се прекине врската помеѓу избришаната дестинациска инстанца `order` и извornата инстанца `customer`, па затоа, пред да се отстрани `order` се поставува `order.customer` на вредност `null`. Изворниот код за правилно отстранување на ентите е:

```
1 order.setCustomer(null);
2 customer.getOrders().remove(order);
```

И во овој случај важат истите правила за отстранување на нереференцирани ентитети со поставување на аргументот `orphanRemoval`.

Во оригиналната дефиниција на ентитетот `Order` во изворниот код [5.12](#) не е наведен типот на каскада, па затоа било која промена на состојбата на неговите инстанци не се пренесува на инстанцата од ентитетот `Customer` со која е во релација. Секако, доколу е потребно, може да се дефинира однесувањето и на насоката повеќе-кон-еден дефинирана во `Order` преку задавање вредности на аргументот `cascade`.

Сите споменати правила за дефиниција на каскадите и референцирањето на зависни ентитети важат и кај релациите еден-кон-еден и повеќе-кон-повеќе.

5.7.3 Вежба: работа со ентитети во релација

Со цел да разгледаме примери од работа со инстанци на ентитети кои се во релација, ќе го прошириме претходниот пример, фокусирајќи се на веќе користените класи на ентитетите `Customer` и `Order` и ќе ги менуваме насоките и начините на однесување на нивните релации.

Најпрво, во пакетот `model` поставете ја содржината на класите `Customer` и `Order` според кодот даден во [5.31](#) и [5.32](#), соодветно.

```

1  @Getter
2  @Setter
3  @NoArgsConstructor
4  @RequiredArgsConstructor
5  @Entity
6  public class Customer {
7      @Id @GeneratedValue
8      private Integer id;
9      @NotNull
10     private String firstName;
11     @NotNull
12     private String lastName;
13     @OneToMany(cascade=CascadeType.ALL)
14     @JoinColumn(name = "customer_id")
15     private List<Order> orders = new ArrayList<>();
16     public String toString(){
17         return "id: " + id + ", firstName: " + firstName + ", lastName: " +
18             firstName +
19             ", orders: [" + orders.stream().map(Object::toString) .
collect(Collectors.joining(", "))+""]";

```

```
20    }
21 }
```

Изворен код 5.31: Содржина на изворен ентитет `Customer` на еднонасочна релација еден-кон-повеќе

```
1 @Getter
2 @Setter
3 @NoArgsConstructor
4 @RequiredArgsConstructor
5 @Entity
6 @Table(name = "eshop_order")
7 public class Order {
8     @Id @GeneratedValue
9     private Integer id;
10    private Date date = new Date();
11    @NotNull
12    private double amount;
13    private OrderStatus status = OrderStatus.INITIALIZED;
14    public String toString(){
15        return "id: "+ id + ", amount: "+ amount + ", status: "+ status;
16    }
17 }
```

Изворен код 5.32: Содржина на дестинациски ентитет `Order` на релација еден-кон-повеќе

Во пакетот `repository` креирајте репозиториуми за CRUD операции со овие ентитети со име `CustomerRepository` и `OrderRepository` како во извornите кодови [5.33](#) и [5.34](#).

```
1 @Repository
2 public class CustomerRepository {
3     @PersistenceContext
4     EntityManager em;
5     @Transactional
6     public Customer findById(Integer id){
7         return em.find(Customer.class,id);
8     }
9     @Transactional
10    public Customer create(Customer customer){
11        em.persist(customer);
12        return customer;
```

```

13 }
14 @Transactional
15 public Customer update(Customer customer){
16     return em.merge(customer);
17 }
18 @Transactional
19 public void delete(Integer id){
20     Customer customer = em.find(Customer.class,id);
21     em.remove(customer);
22 }
23 @Transactional
24 public List<Customer> findAll(){
25     TypedQuery<Customer> query = em.createQuery("select c from Customer
26         c", Customer.class);
27     return query.getResultList();
28 }

```

Изворен код 5.33: Содржина на репозиториумот CustomerRepository за работа со ентитетот Customer

```

1 @Repository
2 public class OrderRepository {
3     @PersistenceContext
4     EntityManager em;
5     @Transactional
6     public Order findById(Integer id){
7         return em.find(Order.class,id);
8     }
9     @Transactional
10    public List<Order> findAll(){
11        TypedQuery<Order> query = em.createQuery("select o from Order o",
12            Order.class);
13        return query.getResultList();
14    }
15    @Transactional
16    public Order create(Order order){
17        em.persist(order);
18        return order;
19    }
20    @Transactional

```

```

20   public Order update(Order order){
21       return em.merge(order);
22   }
23   @Transactional
24   public void delete(Integer id){
25       Order order = em.find(Order.class,id);
26       em.remove(order);
27   }
28   public List<Order> findByCustomerId(Integer id){
29       TypedQuery<Order> query = em.createQuery("select o from Customer c
30           inner join c.orders o where c.id = :id", Order.class);
31       query.setParameter("id",id);
32       return query.getResultList();
33   }

```

Изворен код 5.34: Содржина на репозиториумот OrderRepository за работа со ентитетот Order

Променете ја содржината на апликациската класа така што ќе се декларираат променливи од типот CustomerRepository и OrderRepository и ќе се прошири конструкторот за автоматски да им се додели вредност преку вметнување на зависност. Во рамките на класата дефинирајте го методот runCustomerOrdersExample() кој ќе се повика преку бинот commandLineRunner. Делумната содржината на апликациската класа е дадена во изворниот код 5.35.

```

1  @SpringBootApplication
2  @EntityScan(basePackages = {"mk.ukim.finki.wp.ch4examples"})
3  public class Ch4ExamplesApplication {
4      final AddressRepository addressRepository;
5      final CustomerRepository customerRepository;
6      final OrderRepository orderRepository;
7      public Ch4ExamplesApplication(AddressRepository addressRepository,
8          CustomerRepository customerRepository, OrderRepository
9          orderRepository) {
10         this.addressRepository = addressRepository;
11         this.customerRepository = customerRepository;
12         this.orderRepository = orderRepository;
13     }
14     public static void main(String[] args) {
15         SpringApplication.run(Ch4ExamplesApplication.class, args);
16     }

```

```

15  @Bean
16  public CommandLineRunner commandLineRunner(ApplicationContext ctx) {
17      return args -> {
18          runCustomerOrderExample();
19      };
20  }
21  public void runCustomerOrderExample(){
22      Customer customer = new Customer("Spring","Developer");
23      Order order = new Order(20.00);
24      customer.getOrders().add(order);
25      order = new Order(100.00);
26      customer.getOrders().add(order);
27      System.out.println("----JPA: Persist customer----");
28      customerRepository.create(customer);
29      System.out.println("----New customer----");
30      System.out.println(customer.toString());
31  }
32 }
```

Изворен код 5.35: Содржина на апликациска класа за работа со ентитетите `Customer` и `Order` во еднонасочна врска еден-кон-повеќе

Откако ќе се изврши апликацијата, најпрво се креира инстанца на изворниот ентитет `Customer`, а потоа уште две инстанци на ентитетот `Order` кои се назначуваат како дестинациски ентитети. Поради каскадата на релацијата `@OneToOne` од типот `CascadeType.ALL`, во моментот кога ќе се повика методот `customerRepository.create(customer)`, се перзистираат сите инстанци што се дел од релација. Тоа може да се заклучи според испечатените вредности на конзолата:

```

1  ----New customer----
2  Customer(id: 1 firstName: Spring, lastName: Spring, orders:
3      [Order(id: 2, amount: 20.0, status: INITIALIZED),
4      Order(id: 3, amount: 100.0, status: INITIALIZED)])
```

За да добиеме појасна слика што се случува во базата на податоци, ќе ги погледнеме SQL барањата кои ги генерирал `EntityManager`:

```

1  ----JPA: Persist customer----
2  Hibernate: select nextval ('hibernate_sequence')
3  Hibernate: select nextval ('hibernate_sequence')
4  Hibernate: select nextval ('hibernate_sequence')
5  Hibernate: insert into customer (first_name, last_name, id) values (?, ?, ?)
6  Hibernate: insert into eshop_order (amount, status, id) values (?, ?, ?)
```

```

7 Hibernate: insert into eshop_order (amount, status, id) values (?, ?, ?)
8 Hibernate: update eshop_order set customer_id=? where id=?
9 Hibernate: update eshop_order set customer_id=? where id=?

```

Испечатените наредби укажуваат дека Hibernate најпрво креира три секвентни броеви кои ќе ги искористи како идентификатори за внесување на трите инстанци, потоа внесува една редица во табелата `customer` и две редици во табелата `eshop_order`. До овој момент, колонтата со улога на надворешен клуч `customer_id` во табелата `eshop_order` има вредност `null`, што значи дека не постои никава релација помеѓу редиците за нарачка и клиент. За да се оствари релацијата, Hibernate повикува уште две наредби со кои ја ажурира табелата `eshop_order` и како вредност на надворешниот клуч го поставува идентификаторот `id` на клиентот кој ги поседува тие нарачки.

Клучна поставка за да функционира овој код е постапување на каскада кај релацијата `@OneToMany` на вредност `CascadeType.PERSIST` или, како во нашиот случај, на вредност `CascadeType.ALL`. Со ова Hibernate откако ќе ја промени состојбата на инстанцата `customer` од `TRANSIENT` во `MANAGED`, истатата промена ја пренесува и на инстанците на `Order` кои се во релација со `customer`. Ако се остави предефинираното однесување на релацијата без да се назначи еден од споменатите типови на каскада, тогаш програмата ќе исфрли грешка која ќе укаже дека има обид на перзистираната инстанца `customer` да ѝ се доделат транзитни инстанци од типот `Order`.

Дека секој повик на метод од `EntityManager` со аргумент `Customer` се пренесува и кај `Order`, може да се заклучи ако во методот `runCustomerOrdersExample1()` ги додадеме следните линии код:

```

1     order = customer.getOrders().get(0);
2     order.setStatus(OrderStatus.PAYED);
3     System.out.println("----JPA: Merge customer----");
4     customerRepository.update(customer);
5     System.out.println("----Updated customer orders----");
6     System.out.println(orderRepository.findByCustomerId(customer.getId()));
7
8     System.out.println("----JPA: Remove customer----");
9     customerRepository.delete(customer.getId());
10    System.out.println("----Remaining orders----");
11    System.out.println(orderRepository.findAll());

```

Во кодот, најпрво се зема нарачката со индекс 0 (првата нарачка) и се менува нејзиниот статус. Потоа се повикува методот `customerRepository.update(customer)` кој во позадина повикува

`em.merge(customer)`. Иако експлицитно не се повикува `em.merge(order)`, поради каскадата од типот `CascadeType.ALL` повикот на методот `merge()` се пренесува и на сите инстанци од `Order` кои се во релација со `customer`. Ако се повика методот `(orderRepository.findByCustomerId(customer.getId()))` за да се преземат нарачките од базата за конкретниот корисник, се добива листата од нарачки во кои првата нарачка има променет статус со вредност `PAYED`.

```
1 ----Updated customer orders----
2 [Order(id=2, amount=20.0, status=PAYED),
3 Order(id=3, amount=100.0, status=INITIALIZED)]
```

За да се изврши операцијата на ажурирање на податоците и листање на сите нарачки, `EntityManager` ги генерира следните SQL барањата:

```
1 ----Merge customer-----
2 Hibernate: select ... from customer customer0_
3     left outer join eshop_order orders1_
4         on customer0_.id=orders1_.customer_id
5         where customer0_.id=?
6 Hibernate: update eshop_order set amount=?, status=? where id=?
```

Најпрво се извршува `select` барање од табелите `customer` и `order` споени преку `join` кое ги повлекува последните вредности на клиентот `customer` од табелата `customer` и вредностите на нарачката `order` која се уредува. Потоа се извршува `update` за да се ажурираат вредностите на редицата во табелата `eshop_order` за конкретната нарачка со новите вредности од нејзината инстанца `order`. Во овој случај, Hibernate детектира дека има промена само кај една нарачка, па затоа генерира само една `update` наредба. Во случај да има промена на својствата и на `customer` и на втората нарачка, тогаш ќе генерира две наредби `update`: една за ажурирање на табелата `eshop_order` и една `update` за ажурирање на табелата `customer`.

Бо последните линии код на методот `runCustomerOrdersExample1()` се брише инстанцата `customer` преку повик на методот `customerRepository.delete(customer.getId())` кој во имплементацијата го повикува методот `em.remove(id)`. Поради каскадата од типот `CascadeType.ALL` повикот на методот `remove()` се пренесува и на сите инстанци од `Order` кои се во релација со `customer` што резултира со нивно бришење. Со оглед на тоа што двете нарачки во примерот се единствените нарачки во табелата, методите `(orderRepository.findByCustomerId(customer.getId()))` `(orderRepository.findAll())` враќаат празна листа.

```
1 ----Customer orders----
2 []
```

```
3 ---Remaining orders---
```

```
4 []
```

За бришење на податоците за клиентот и неговите нарачки, EntityManager ги генерира следните SQL барањата:

```
1 ----JPA: Remove customer----
```

```
2 Hibernate: select ...
```

```
3   from customer customer0_
```

```
4     where customer0_.id=?
```

```
5 Hibernate:select ... from eshop_order orders0_ where orders0_.customer_id=?
```

```
6 Hibernate: delete from eshop_order where id=?
```

```
7 Hibernate: delete from eshop_order where id=?
```

```
8 Hibernate: delete from customer where id=?
```

Hibernate најпрво генерира `select` барање за клиент со идентификатор кој одговара на `customer.id`. За потсетување, во имплементацијата на методот `customerRepository.delete(int id)` во изворниот код на репозиториумот [5.33](#), прво се повикува `em.find(Customer.class, id)` за да се внесе ентитетот во перзистентниот контекст за потоа да може да се избрише. Токму овој повик резултира со генерирање на првата `select` наредба. Втората `select` наредба се извршува за да се добијат сите инстанци на ентитетот `Order` кои се во релација со `customer` за да се внесат во перзистентниот контекст и подоцна да се избришат. На крајот се бришат нарачките за клиентот преку две `delete` наредби за `eshop_order`, а потоа се брише и самиот клиент преку `delete` за табелата `customer`.

Ако го промениме типот на каскада на релацијата еден-кон-повеќе така што ќе ја поставите вредноста `CascadeType.PERSIST` како во кодот:

```
1 @OneToMany(cascade=CascadeType.PERSIST)
2   @JoinColumn(name = "customer_id")
3   private List<Order> orders = new ArrayList<>();
```

и потоа повторно ја извршиме истата апликација така што ќе се повикаат истите методи за работа со ентитети од методот `runCustomerOrdersExample1()`. Добиениот излез е следен:

```
1 ----New customer-----
```

```
2 Customer(id=1, firstName=Spring, lastName=Developer,
3   orders=[Order(id=2, amount=20.0, status=INITIALIZED),
4     Order(id=3, amount=100.0, status=INITIALIZED)])
```

```
5 ----Updated customer orders-----
```

```
6 [Order(id=2, amount=20.0, status=INITIALIZED),
7   Order(id=3, amount=100.0, status=INITIALIZED)]
```

```
8 ----Customer orders----
```

```

9  []
10 ----Remaining orders-----
11 [Order(id=2, amount=20.0, status=INITIALIZED),
12 Order(id=3, amount=100.0, status=INITIALIZED)]

```

Од добиените резултати може да се заклучи дека типот на каскада `CascadeType.PERSIST` дозволува при внесување во перзистентниот контекст на `customer` да се пренесе методот `persist()` од `customer` на двете додадени нарачки, но не дозволува да се пренесат методите `merge()` и `remove()`, па затоа, и покрај тоа што направивме промена на статусот на првата нарачка, таа промена не се синхронизираше во базата како што беше во претходниот случај. Исто така, и покрај тоа што го избришавме `customer`, неговите нарачки останаа неизбришани. Како што видовме претходно, ентиетот `Order` е сопственик на релацијата, па затоа во неговата табела `eshop_order` постои надворешен клуч `customer_id`. Поради бришењето на клиентот со кој овие нарачки беа порврзани, надворешниот клуч се ажурира на вредност `null`, па затоа нарачките продолжуваат да егзистираат во табелата. Тоа може да се види од резултатот кој го враќа методот `orderRepository.findAll()`. Од друга страна, методот кој ги враќа сите нарачки за избришаниот клиент враќа празна листа.

SQL барањата за каскада од типот `CascadeType.PERSIST` генериирани од `EntityManager` са следните:

```

1  ----JPA: Persist customer----
2  ... insert commands...
3  ----JPA: Merge customer----
4  Hibernate: select ... from customer customer0_ where customer0_.id=?
5  Hibernate: select ... from eshop_order orders0_ where orders0_.customer_id=?
6
7  ----JPA: Remove customer----
8  Hibernate: select ... from customer customer0_ where customer0_.id=?
9  Hibernate: update eshop_order set customer_id=null where customer_id=?
10 Hibernate: delete from customer where id=?

```

Најрпво се извршуваат истите `insert` наредби за внесување на еден клиент и две нарачки (намерно изоставени во содржината на конзолата). Кога се повикува `orderRepository.update(customer)`, само се повлекуваат последните верзии на ставките од базата на податоци, но не се повикува `update`. Во случај да имаше промени во својствата на `customer` ќе беше генерирана `update` наредба само за табелата `customer`, но во ниеден случај не се генерира `update` за табелата `eshop_user`. При повик на `orderRepository.delete(customer.getId())` се генерира една `select` наредба за внесување на `customer` во презистентниот контекст. Потоа, преку `update` се поставува надворешниот клуч `customer_id` во табелата `eshop_order` на вред-

ност `null` и на крајот се брише редицата од табелата `customer` која одговара на ентитетот `customer`. Во ова сценарио, двете редици кои одговараат на веќе избришаната нарачка не се референцирани од ниту еден клиент и стануваат сираци, па за базата да не се полни со непотребни податоци, потребно е или програмерот рачно да ги избира или да се користи опцијата `orphanRemoval=true` при дефинирање на релацијата `@OneToMany` во `Customer`.

Ако ја променим релацијата `@OneToMany` на тој начин што ќе го промените предефинираното однесување во однос на справување со нереференцирани инстанци поставувајќи `orphanRemoval=true` и извршете ја апликацијата. Во тој случај, на конзолата ќе биде испишано:

```

1 ----New customer-----
2 Customer(id=1, firstName=Spring, lastName=Developer,
3   orders=[Order(id=2, amount=20.0, status=INITIALIZED),
4           Order(id=3, amount=100.0, status=INITIALIZED)])
5 ----Updated customer orders-----
6 []
7 ----Remaining orders-----
8 []

```

Од добиените резултати можеме да заклучиме дека откако `order` е отстранета од врската со `customer`, но не е експлицитно избришана преку `remove()`, `EntityManager` ја брише извршувајќи `delete` наредба за табелата `eshop_order`, бидејќи останува нерференцирана. Наредбите кои се извршуваат во овој случај се:

```

1 ----JPA: Remove customer----
2 Hibernate: select ... from eshop_order orders0_ where orders0_.customer_id=?
3 Hibernate: update eshop_order set customer_id=null where customer_id=?
4 Hibernate: delete from eshop_order where id=?
5 Hibernate: delete from eshop_order where id=?
6 Hibernate: delete from customer where id=?

```

Од испишаните SQL наредби може да се види дека најпрво се ажурира надворешниот клуч на нарачките со што тие стануваат нереференцирани, а потоа истите се отстрануваат од табелата `eshop_order`. На крајот, се брише и самиот купец. Иако никаде експлицитно не се повика методот `remove()` за ентитетот `Order`, `EntityManager` ги брише овие ставки поради поставката да се отстранат нереференцираните дестинациски ентитети.

Наредно, ќе креираме релација повеќе-кон-еден од `Order` кон `Customer` со цел да ја дефинирате нивната релација како двонасочна. За таа цел, во содржината на класата `Customer` ќе ја променим анатацијата `@OneToMany` на својството `orders`

според слендиот код:

```
1  @OneToMany(mappedBy = "customer", cascade=CascadeType.ALL, orphanRemoval
2      = true)
3  private List<Order> orders = new ArrayList<>();
```

а во класата `Order` додате ја другата страна од релацијата преку изворниот код:

```
1  @ManyToOne
2  @JoinColumn(name = "customer_id")
3  private Customer customer;
```

Следно, ќе креираме метод `runCustomerOrdersExample2()` со следната содржина:

```
1  public void runCustomerOrderExample2(){
2      Customer customer = new Customer("Spring", "Developer");
3      Order order = new Order(20.00);
4      customer.getOrders().add(order);
5      order.setCustomer(customer);
6      order = new Order(100.00);
7      customer.getOrders().add(order);
8      order.setCustomer(customer);
9      System.out.println("----JPA: Persist customer----");
10     customerRepository.create(customer);
11     System.out.println("----New customer----");
12     System.out.println(customer.toString());
13
14     order = customer.getOrders().get(0);
15     customer.getOrders().remove(order);
16     order.setCustomer(null);
17
18     System.out.println("----JPA: Merge customer with removed order----");
19     customerRepository.update(customer);
20     System.out.println("----All orders----");
21     System.out.println(orderRepository.findAll());
22 }
```

Кодот иницијално ги внесува истите инстанци како и во претходните примери, со таа разлика што при додавањето или бришењето на ставка во листата `orders` во ентитет `Customer`, се ажурираат и референците во обратната насока од `Order` кон `Customer`. Во примерот, по внесувањето на неколку нарачки кај клиентот и нивно вметнувањето во перзистентниот контекст, во кодот се отстранува една нарачка `order` од истата листа `orders` на ентитетот `customer` и истиот се перзишира. Потоа се печатат сите постоечки нарачки. Ако се разгледаат пораките испишани на

конзолата, ќе се забележи дека ентитетите се успешнио внесени во базата, а потоа селектираната нарачка е успешнио избришана.

```

1 ----New customer-----
2 Customer(id=1, firstName=Spring, lastName=Developer,
3   orders=[Order(id=2, amount=20.0, status=INITIALIZED),
4           Order(id=3, amount=100.0, status=INITIALIZED)])
5 ----Customer orders-----
6 Order(id=3, amount=100.0, status=INITIALIZED)]
7 ----All orders-----
8 Order(id=3, amount=100.0, status=INITIALIZED)]
```

Она што ја прави двонасочната врска поразлична се SQL барањата кои се испраќаат кон базата. Во овој случај, `EntityManager` ги генерира следните барања:

```

1 ----JPA: Persist customer-----
2 ...generate sequence numbers...
3 Hibernate: insert into customer (first_name, last_name, id) values (?, ?, ?)
4 Hibernate: insert into eshop_order (amount, customer_id, date, status, id)
5   values (?, ?, ?, ?, ?)
5 Hibernate: insert into eshop_order (amount, customer_id, date, status, id)
6   values (?, ?, ?, ?, ?)
6 ----JPA: Merge customer with removed order-----
7 Hibernate: select ...
8 Hibernate: delete from eshop_order where id=?
```

За разлика од еднонасочната релација каде најпрво се внесуваа клиент и две нарачки, а дури потоа се прави ажурирање на нарачките за да го референцираат клиентот, кај двонасочната релација тоа се прави уште при самото внесување на нарачките. Остранивањето на нарачката од клиентот резултира со нејзино бришење од базата поради поставката `orphanRemoval=true`.

Поради двонасочната природа на реализацијата помеѓу ентитетите, внесувањето и додавањето бара дополнително внимание при ажурирање на референциите. Вавката релација е покомплексна и има полоши перформанси, па затоа се препорачува користење на едноасочна релација освен во случаите каде тоа е неопходно.

5.8 Spring Data JPA

Во сите досгашни примери за работа со ентитети беше потребно да се креираат посебни класи т.н. репозиториуми во кои беа сместени методите за CRUD

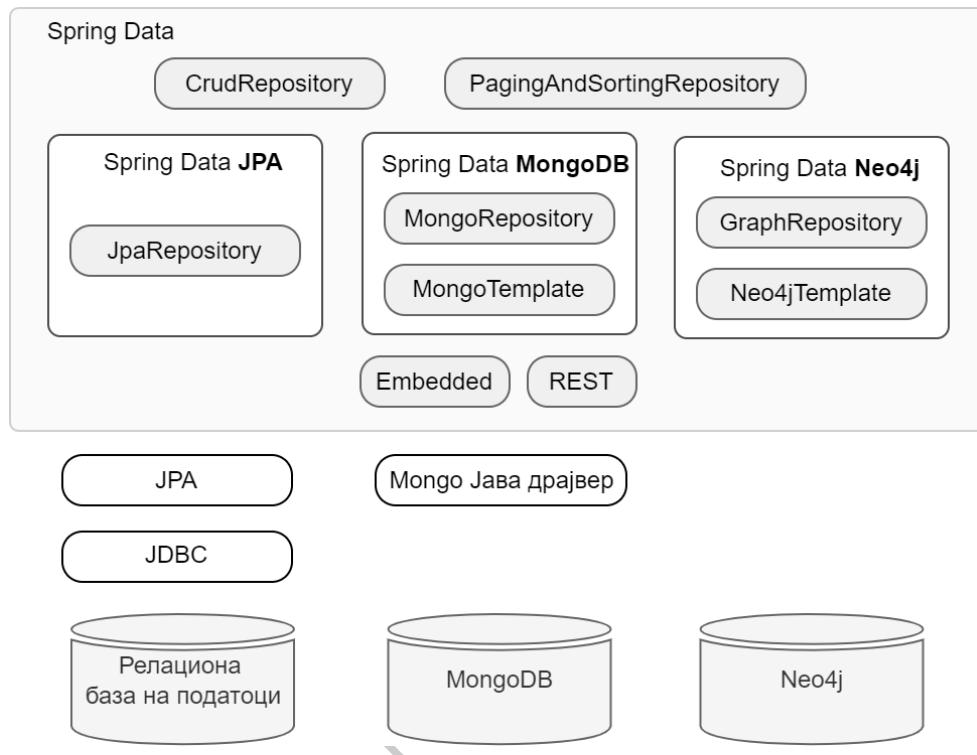
и други дополнителни операции за ентитетите имплементирани преку методите кои ги нуди EntityManager и Query. Ако повторно се навратиме и ги разгледаме, ќе заклуччиме дека голем дел од кодот се повторува бидејќи за секој ентитет се потребни истите базични CRUD операции чија имплементација единственото се разликува по ентитетите врз кои се извршуваат тие операции. На пример, AddressRepository и CustomerRepository дефинираат метод за зачувување на нова инстанца `public Address create(Address address)` и `public Customer create(Customer customer)` кои интерно ги повикуваат методите `em.persist(address)` и `em.persist(customer)`, соодветно. Дополнително, за секоја специфична операција која се разликува од стандардните CRUD операции и е тесно поврзана со самите својства на ентитетите, како што е операцијата на пребарување на ентитети кои задоволуваат одредени услови, потребно е да се пишуваат JPQL или SQL наредби во Query објектите. Ваквиот начин на работа налага дуплирање на голем дел од кодот, што го прави програмирањето на податочниот слој помалку ефикасно и подложно на грешки, особено кај покомплексни апликации кои се состојат од голем број на ентитети.

Spring Data е проект на високо ниво го адресира и решава проблемот на работа со податочниот слој, независно за каков тип на податочно складиште станува збор. Содржи широка палета на подпроекти кои го решаваат проблемот за тековно најкористените податочни складишта, меѓу кои се релационите или нерелационите бази на податоци. Главната придобивка од неговото користење е што не треба рачно да го пишуваме целиот код за репозиториумот со CRUD операции и операции за пребарување на податоци, па затоа ја поедноставува работата со податочниот слој. Во зависност од технологијата на податочното складиште, на пример JPA за релациони бази на податоци, MongoDB за нерелациони бази на податоци или Neo4J за граф-базирани бази на податоци, постојат различни модули кои нудат и дополнителни функционалности специфични за нив. Со оглед на тоа што во оваа книга користиме објектно-ориентиран пристап на работа со реалциони бази преку JPA спецификацијата, главниот фокус ќе биде насочен на Spring Data JPA.

На сликата 5-15 е прикажан преглед на Spring Data и дел од модулите кои го сочинуваат, вклучувајќи го Spring Data JPA. Главниот проект Spring Data се состои од генерички интерфејси за репозиториумите:

- `Repository` кој е највисока абстракција со која се означува дека Spring Data ќе биде задолжен за овозможување на имплементација на интерфејсот
- `CrudRepository` преку кој нуди имплементација на CRUD и останати специфични операции врз објектите на ентитетите
- `PagingAndSortingRepository` преку кој нуди имплементација на опции за странице и подредување на ентитетите при пребарување.

За секој од поддржаните технологии на податични складишта, Spring Data нуди соодветна имплементација на овие интерфејси. Исто така, за секоја од технологиите, Spring Data ги поддржува и анотациите за дефинирање на доменските класи и релациите помеѓу нив во случај тие да се поддржани. На пример, Spring Data ги поддржува познатите JPA анотации за ентитети (пр. `@Entity`, `@Column` итн.) но и за дефинирање на релации помеѓу нив (пр. `@OneToMany`, `@ManyToOne` итн.). Освен



Слика 5-15: Преглед на Spring Data модули

основите унифицирани операции, за секој тип на податично складиште нуди и модули кои поддржуваат операции специфични за него. Така, модулот Spring Data JPA ги нуди следните функционалности:

- интерфејс за репозиториумот `JpaRepository` кој се користи за имплементирање на специфични функционалности за JPA кои не се карактеристични за останатите складишта (на пример, `flush()`).
- можност за креирање на специфичните барања за работа со ентитетите само преку именување на методите во интерфејсите на репозиториумите според посебна конвенција за именување

Според сликата 5-15, доколку се користи за работа со релациони бази на податоци, Spring Data целосно се потпира на спецификациите на JPA како слој под него. Тоа значи дека секој од методите кои автоматски ги генерира како резултат на користење на наведените репозиториуми. Овде се користи имплементација

базирана на правилата дефинирани во JPA спецификацијата. Следствено, во зависност од избраниот JPA провјадер, се генерираат соодветни JDBC наредби кои се испраќаат до релационата база на податоци како чисти SQL барања.

5.8.1 Интерфејси за репозиториуми

Spring Data овозможува целосно автоматизирано генерирање на методи за напредни CRUD операции со ентитетите во податочниот слој преку интерфејсот `CrudRepository` со следната дефиниција:

```

1 public interface CrudRepository<T, ID> extends Repository<T, ID> {
2     <S extends T> S save(S entity);
3     Optional<T> findById(ID primaryKey);
4     Iterable<T> findAll();
5     long count();
6     void delete(T entity);
7     boolean existsById(ID primaryKey);
8 }
```

Интерфејсот е проширување на основниот интерфејс `Repository<T, ID>` дефиниран со типот на класта на ентитетот `T` на кој се однесува и типот `ID` на идентификаторот на тој ентитет. За секој ентитет се дефинира репозиториуми кој е проширување на `CrudRepository` и во чија дефиниција се наведува конкретниот тип, односно класа на ентитетот. Според тоа, `T` преставува генерички тип и при компајлирање се заменува со типот на конкретната класа на ентитет за која се дефинира репозиториумот.

Основен интерфејс `Repository<T, ID>` служи само за маркирање на типовите на податоци. Од друга страна, интерфејсот `CrudRepository` кој исто така се дефинира преку типот на ентитетот и типот на неговиот идентификатор содржи потпис за следните CRUD методии:

- `S save(S entity)`: зачувува ентитетот од тип `S`, кој може да биде проширување на типот `T`, т.е. `S extends T`.
- `Optional<T> findById(ID id)` пребарува ентитет со идентификатор од тип `ID` и враќа `Optional` инстанца која може, но и не мора, да има вредност. Ако има вредност (постои бараниот ентитет) тогаш `isPresent()` повикан врз објектот враќа вредност `true`, а методот `get()` го враќа ентитетот `T`. Ако не посоди вредност, тогаш со методот `orElse()` може да се испрати произволна вредност (на пример `null` или празна истанца од `T`)
- `List<T> findAll()` враќа листа од сите ентитети од типот `T`
- `Long count()` го враќа бројот на сите ентитети од типот `T`
- `void delete(T entity)` го брише ентитетот `T`

- Boolean `existsById(ID id)` враќа `true` ако постои ентитет со идентификатор `T`

За да Spring Data генерира код за имплементација на секој од методите на интерфејсот `CrudRepository` за конкретен ентитет `Customer`, потребно е да се дефинира репозиториум, на пример `CustomerRepository`, во формат на интерфејс кој ќе го проширува интерфејсот `CrudRepository` на следниот начин:

```
1 interface CustomerRepository extends CrudRepository<Customer, Long>{
2 }
```

Со ваквата дефиниција, сите методи дефинирани во `CrudRepository` стануваат дел и од `CustomerRepository`, но овој пат, тие методи ќе се однесуваат на конкретен тип на ентитет `Customer` чиј индентификатор е од типот `Long`.

Како и секоја класа која имплементира некој интерфејс, така и класата во проектот со улога на репозиториум треба да ги имплементира сите методи од интерфејсот `CustomerRepository`. Но, за разлика од претходните примери каде тоа го правевме мануелно, со помош на Spring Data, преку вметнување на променлива од типот `CustomerRepository` во позадина се генерира инстанца на класа со код кој веќе содржи имплементација на секој од методите на неговиот потпис. Ако проектот користи JPA спецификација, тогаш за имплементација на методите од интерфејсот ќе се користат методите на `EntityManager` и `Query`.

На пример, ако се креира репозиториум, односно интерфејс за ентитетот `Order`

```
1 interface OrderRepository extends CrudRepository<Order, Integer>{
2 }
```

тогаш за методот `save()` се креира следниот имплементациски код:

```
1 @Transactional
2 public Order save(Order order){
3     if (order.getId() == null){
4         em.persist(order);
5         return order;
6     } else
7         return em.merge(order);
8 }
```

Овој метод е замена за двета методи `create()` и `update()` кои ги дефиниравме во изворниот код [5.34](#). Во зависност од тоа дали ентитетот кој треба да се зачува е во перзистентниот контекст или не, `save()` го повикува методот `persist()` или `merge()`.

Важно е да се забележи дека при имплементацијата, секој метод е назначен

со анотацијата `@Transactional`, а самата класа која ќе се креира од интерфејсот е означена со анотацијата `@Repository`.

На сличен начин, Spring Data ги креира имплементациите на сите останати методи од интерфејсот `CrudRepository` и со тоа значително го олеснува и забрзува работењето со податочниот слој, елиминирајќи ја потребата самите програмери да ги имплементираат методите за CRUD операции.

Spring Data нуди и уште еден понапреден интерфејс `PagingAndSortingRepository` кој е проширување на `CrudRepository`. Покрај основните CRUD функционалности наследени од `CrudRepository`, интерфејсот нуди сортирање и страничење на податоците кои се добиваат при читање на листи од ентитети. Неговата дефиниција е:

```

1 public interface PagingAndSortingRepository<T, ID> extends
2   CrudRepository<T, ID> {
3   Iterable<T> findAll(Sort sort);
4   Page<T> findAll(Pageable pageable);
5 }
```

Методот `findAll(Sort sort)` враќа итеративна листа од ентитети и користи аргумент од класата `Sort` кој овозможува специфицирање на:

- насока на подредување (`Sort.Direction.ASC` или `Sort.Direction.DESC`) и
- својство на ентитетот според кое ќе се подредат ентитетите.

На пример, ако станува збор за репозиториум дефиниран за ентитетот `Order`, тогаш `findAll(Sort.by(Sort.Direction.DESC, "date"))` ќе ги врати сите нарачки во опаѓачки редослед, подредени според датумот на креирање (најновата нарачака ќе биде прва во листата, а најстарата последна).

Методот `findAll(Pageable pageable)` враќа објект од класата `Page` кој преставува само една фракција од сите ентитети. За да се добие листа од ентитетите кои се дел од таа страница се повикува методот `getContent()`. Класата содржи и дополнителни корисни методи како што се `getTotalElements()` и `getTotalPages()` кои ги даваат вкупниот број на ентитети од пребарувањето и вкупниот број на страници во кои се сместени тие ентитети. Методот користи аргумент од класата `Pageable` кој овозможува специфицирање на:

- големина на страница, односно максимален број на ентитети кои ги содржи,
- реден број на страница и
- подредување на елементите преку класата `Sort`

На пример, ако станува збор за репозиториум дефиниран за ентитетот `Order`, тогаш `findAll(PageRequest.of(1, 10, Sort.by(Sort.Direction.DESC, "date")))` ќе ја врати втората страница од последни нарачки подредени во опаѓачки редослед според датумот на креирање чија големина е 10 нарачки. Ако има вкупно 35 нарачки подредени по својството `date`, тогаш тие може да се сместат во вкупно 4 страници со големина од по 10 елементи (последната страница ќе содржи само 5 елементи), а конкретната страница со индекс 1 која ќе ја врати методот ќе ги содржи елементите со индекс од 10 до 19, индексирани според подредувањето.

За понапредни функционалности специфични за JPA спецификацијата може да се користи интерефјост `JpaRepository` кој е дел од Spring Data JPA и е проширување на `PagingAndSortingCrudRepository`. Неговата дефиниција е:

```

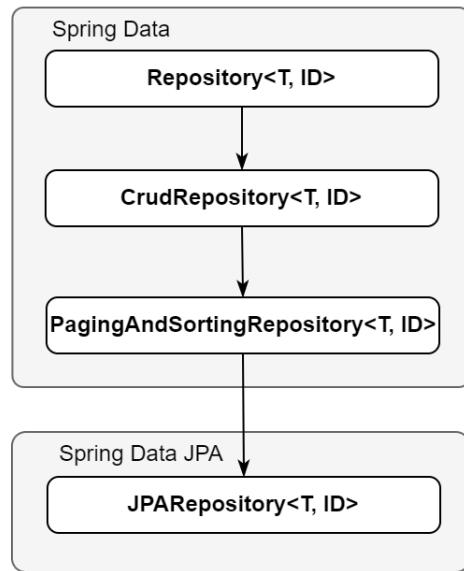
1 public interface JpaRepository<T, ID> extends PagingAndSortingRepository<T,
2     ID> {
3     List<T> findAll();
4     List<T> findAll(Sort sort);
5     List<T> findAllById(Iterable<ID> ids);
6     void flush();
7     <S extends T> List<S> saveAll(Iterable<S> entities);
8     <S extends T> saveAndFlush(S entity);
9     <S extends T> List<S> saveAllAndFlush(Iterable<S> entities);
10    void deleteInBatch(Iterable<T> entities);
11    ...
12 }
```

Овој тип на репозиториум нуди методи за пребарување кои врќаат ентитет во објект од типот `List` (не `Iterable` како претходно), пребарување на повеќе ентитети според листа од идентификатори, инстантно синхронизирање на перзистениот контекст, зачувување на листа од ентитети, зачувување на единствен ентитет или листа од ентитети и нивно инстантно синхронизирање со базата, бришење на повеќе ентитети одеднаш итн.

На сликата 5-16 е даден преглед на Spring Data и Spring Data JPA репозиториумите и нивната хиерархиска подреденост. Според дефиницијата на репозиториумите кои ги разгледавме до сега, `JpaRepository` нуди најголем број функционалности, но сепак, `CrudRepository` и `PagingAndSortingRepository` најчесто ги задоволуваат потребите за имплементација на функционалностите на апликациите.

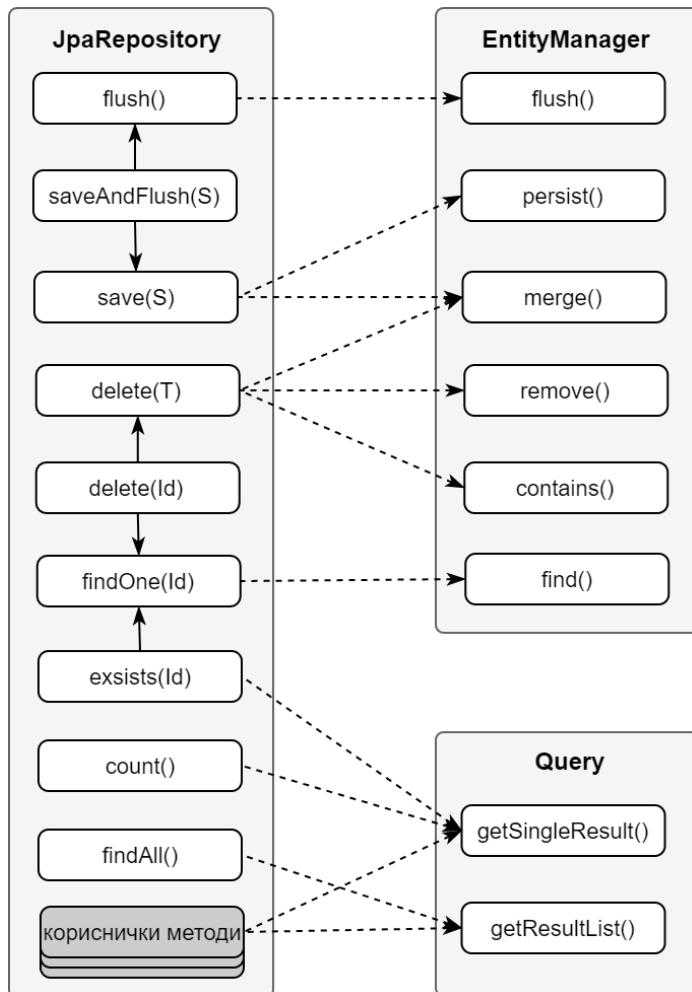
На сликата 5-17 е дадена споредба помеѓу имплементациите на дел од методите на репозиториумот `JpaRepository` и методите од JPA спецификацијата.

Според сликата, имплементацијата на секој од методите е сленда:



Слика 5-16: Преглед на репозиториуми

- `flush()` го повикува истоимениот метод во JPA за да го синхронизира ментално перзистентниот контекст
- `save(S)` го повикува методот `persist()` ако ентитетот е транзитен (нов) или `merge()` ако ентитетот е откачен. По повикот на методите, ентитетот станува перзистентен и се синхронизира откако ќе се извршат сите операции од трансакцијата
- `saveAndFlush(S)` ги повикува `save()` и `flush()` од самиот репозиториум
- `delete(T)` најпрво го повикува `contains()` за да провери дали ентитетот кој треба да се избрише е во перзистентниот контекст. Ако не е, односно ако е во откачена состојба, се внесува во перзистентниот контекст со `merge()`, а потоа се означува за бришење со `remove()`. Перзистентните објекти веднаш се отстрануваат со `remove()`
- `findById(id)` го повикува `find()` наведувајќи ја класата на ентитетот и неговиот идентификатор.
- `delete(id)` најпрво го сместува ентитетот во база преку `findById(id)`, а потоа го повикува `delete(T)` за да го избрише
- `exists(id)` го повикува `findById(id)` и во зависност од тоа дали ќе врати инстанца или `null` враќа соодветна `boolean` вредност
- `count()` го повикува методот `getSingleResult()` од `TypedQuery` на кое се пренесува соодветна JPQL наредба за да се пребројат сите ентитети и да се добие единствена вредност. Истиот метод се повикува за сите ниостанати кориснички дефинира барување кои треба да вратат единствена вредност
- `findAll()`, `findBy...()` и сите останати кориснички дефинирани бараќа



Слика 5-17: Споредба на методи од JpaRepository и JPA

кои враќаат листа од ентитети го повикуваат методот `getResultList()` на `TypedQuery`

- `deleteBy...()` и сите останати кориснички дефинирани барања кои прават модификација на податоците го повикуваат методот `updateResult()` од `TypedQuery`

Методите за читање се имплементирани со анотација `@Transactional(readOnly=true)`, додека останатите методи кои прават модификација се анотирани само со `@Transactional`, чија предефинирана вредност на `readOnly=false`, што онзначува дека преку таа трансакција може да се извршуваат модификации врз податоците во базата на податоци.

5.8.2 Конвенции за именување на методи

Иако репозиториумите значително го намалуваат бројот на линии код кои треба да се напишат за имплементација на CRUD операциите, при развивање на апликации, потребна е имплементација на специфични барања кои зависат од својствата на ентитетите. Таков пример веќе видовме во репозиториумот `OrderRepository` во изворниот код [5.34](#) каде за методот `findById(CustomerId id)` користевме Query со JPQL синтакса за да се селектираат сите нарачки кои припаѓаат на купец со конкретна вредност на идентификатор `id`:

```

1  @Transactional
2  public List<Order> findById(Integer id){
3      TypedQuery<Order> query = em.createQuery("select o from Order o
4          where o.customer.id = :id", Order.class);
5      query.setParameter("id", id);
6      return query.getResultList();
7  }

```

Spring Data оди уште еден чекор подалеку во олеснување на работата со податочниот слој на тој начин што овозможува дефинирање на методи во интерфејсот со улога на репозиториум со специфична номенклатура за кои потоа креира соодветни имплементации. Со тоа, имплементацијата на специфични барање се сведува само на дефиниција на потпис на методи во интерфејс според точно дефинирани правила, за кои потоа Spring Data креира имплементации користејќи JPQL. Во тој контекст, за имплементација на специфичното пребарување за нарачки во репозиториумот `OrderRepository` доволно е само да се дефинира потписот на методот `findById(CustomerId id)` на следниот начин:

```

1  interface OrderRepository extends CrudRepository<Order, Integer>{
2      public List<Order> findByCustomerId(Integer id);
3  }

```

Според Spring Data, `findBy` означува дека ќе се пребаруваат ентитети по својството кое следи, осноносно својството `customer`. Бидејќи `customer` е ентитет од класата `Customer`, се пребарува по неговото својство `id`. Вредноста на својството за пребарување се пренесува како аргумент на методот и неговиот тип треба да одговара со типот на самото својство во дефиницијата на ентитетот (во случајот `Integer`).

Конвенцијата за именување се состои од следните правила за дефинирање на методи:

- `findBy`, `readBy`, `getBy`, `searchBy` и `queryBy` имаат идентично значење и се користат за пребараување кое враќа листа од ентитети кои го задоволуваат

условот во името

- *countBy* се користати за пребараување кое враќа број на ентитети кои го задоволуваат условот во името
- *deleteBy* и *removeBy* имаат идентично значење и се користат за бришење на ентитети кои го задоволуваат условот во името
- *existsBy* се користи за да се провери дали постојат ентитети кои го задовољуваат условот во името (враќа **boolean** вредност)
- ...*First*<*number*>... и ...*Top*<*number*>... се користат помеѓу *find* (или останатите синоними) и *By* за да се ограничи бројот на резултати кои ги враќа пребарувањето на вредност *number*. На пример, `findFirst10ByName` и `findTop10ByName` ги враќаат првите 10 ставки кои имаат вредност на својството *name* кое се пренесува како аргумент
- ...*Distinct*... се користи помеѓу *find* (или останатите синоними) и *By* за да се вратат само уникатните ентитети кои го испортуваат условот на пребарувањето. На пример, `findDistinctByName` ги враќа само уникатните ентитети кои имаат вредност на својството *name* како на пренесениот аргумент
- После дефинирањето на типот на пребарувањето, следуваат предикати кои ги дефинираат условите. Секој предикат започнува со својство и еден или повеќе клучни зборови за предикати дадени во табелите [5.1](#) и [5.2](#)
- Секој збор кој започнува со голема буква се третира како својство сè до појавување на клучен збор или до крајот на името на методот.
Ако постојат повеќе зборови со голема буква, сите тие се третираат како својство со истото име но со мала почетна буква. На пример, кај `findByCustomerId(Integer id)` се земаат зборовите *CustomerId* и се проверува дали постои својство *customerId*. Ако не постои својство составено од поодделните зборови, се проверува дали постои својство на друг ентитет кој одговара на името на првиот збор со име како и вториот збор, односно дали има својство до кое може да се стигне ако зборовите се одвојат со точка (.). На пример, во класата *Order* не постои својство *customerId*, па затоа се проверува постоењето на својство-ентитет *customer* со својство *id*, односно *customer.id*.
- За да се нагласи експлицитно пребарување според вградено својство, се користи знакот долна црта (_). Во конкретниот пример, тоа би се постигнало преку името `findByCustomer_Id(Integer id)` што би се превело во пребарување по *customer.id*.
- На секое својство во условот одговара аргумент од истиот тип чија позиција зависи од позицијата на својството во името. Ако станува збор за бинарен оператор (пр. *between*) се користат два аргументи

Во табелите [5.1](#) и [5.2](#) се прикажани најчесто користените клучни зборови,

заедно со пример за нивно користење во име на метод придружен со соодветна JPQL репрезентација.

Клучен збор	Пример	JPQL репрезентација
And	findByLastNameAndFirstName	select distinct ... where x.lastName = ?1 and x.firstName = ?2
Or	findByLastNameOrFirstName	... where x.lastName = ?1 or x.firstName = ?2
True, IsTrue	findBySubscribedTrue()	... where x.subscribed = true
False, IsFalse	findBySubscribedFalse()	... where x.subscribed = false
Not	findByNameNot	... where x.name <> ?1
In	findByNameIn(Collection<String> names)	... where x.name in ?1
NotIn	findByNameNotIn(Collection<String> names)	... where x.name not in ?1
Empty, IsEmpty	findByOrdersIsEmpty()	... where x.orders is empty
NotEmpty, IsNotEmpty	findByOrdersNorEmpty()	... where x.orders is not empty
Is, Equals	findByName, findByNameEquals, findByNameIs	... where x.name = ?1
IsNull, Null	findByNameIsNull, findByNameNull	... where x.name is null
IsNotNull, NotNull	findByNameIsNotNull, findByNameNotNull	... where x.name not null
LessThan	findByAmountLessThan	... where x.amount < ?1
LessThanEqual	findByAmountLessThanEqual	... where x.amount <= ?1
Greater Than	findByAmountGreaterThan	... where x.amount > ?1
Greater Than Equal	findByAmountGreaterThanOrEqual	... where x.amount >= ?1

Табела 5.1: Клучни зборови за дефинирање на пребарувања преку имиња на методи во Spring Data

Предикатите можат дополнително да се модифицираат, односно да се подредуваат, да се назначи пребарување на текстуални податоци кое ќе ги третира сите знаци како мали букви или пак ќе бара прецизна споредба итн. Клучните зборови за вакви модификации се дадени во табелата [5.3](#).

Клучен збор	Пример	JPQL репрезентација
Before	findByPurchaseDateBefore	... where x.purchaseDate < ?1
After	findByPurchaseDateAfter	... where x.purchaseDate > ?1
Between	findByPurchaseDateBetween	... where x.purchaseDate between ?1 and ?2
Containing	findByNameContaining	... where x.name like '%?1%'
Like	findByNameLike	... where x.name like ?1
NotLike	findByNameNotLike	... where x.name not like ?1
StartingWith, StartsWith, IsStartingWith	findByNameStartingWith	... where x.name like '?1%'
EndingWith, EndsWith, IsEndingWith	findByNameEndingWith	... where x.name like '%?1'

Табела 5.2: Клучни зборови за дефинирање на пребарувања преку имиња на методи во Spring Data

Клучен збор	Пример	JPQL репрезентација
IgnoreCase, IgnoringCase	findByNameIgnoreCase	... where UPPER(x.name)= UPPER(?1)
AllIgnoreCase, AllIgnoringCase	findByNameAndTitleAllIgnoreCase	where UPPER(x.name)= UPPER(?1) and UPPER(x.title)= UPPER(?2)
OrderBy	findByNameOrderByNameDesc	... where x.name = ?1 order by x.name desc

Табела 5.3: Клучни зборови за модифицирање на пребарувања преку имиња на методи во Spring Data

Сите наредби за пребарување можат да се користат со опцијата за подредување или страничење ако како последен аргумент на методот се пренесе инстанца од класата `Sort` за подредување или `Pageable` за страничење со подредување. Каде страниченето, вредноста која ја враќа методот треба да биде инстанца од класата `Page<T>`.

За изведените методи важат истите правила за имплементација во однос на користење на `@Transactional`, односно барањата за читање можат само да читаат податоци, а оние за модификација можат да модифицираат податоци

преку трансакцијата.

5.8.3 Кориснички дефинирани барања

Комбинирањето на предикати користејќи ги правилата и клучните зборови нуди можност за дефинирање на сложени пребарувања без да се напише ниту една линија JPQL или SQL код. Сепак, постојат ситуации каде е потребно да се дефинираат уште посложени или поспецифични барања како што се поврзување на повеќе табели, агрегација, напредна промена на податоци итн. Овие барања не може да се имплементираат преку име на методите, па затоа Spring Data JPA нуди можност за дефинирање на понапредни пребарувања преку JPQL или SQL. Секој метод за пребараување дефинирани директно преку неговото име може да се замени и со кориснички дефинирано барање.

Кориснички дефинираните барања се креираат на тој начин што пред името на методот се поставува анотацијата `@Query` во која како аргумент се пренесува JPQL наредба. Бидејќи дефиницијата на пребарувањата е специфична за JPA, методите кои ќе користат напредни пребарувања треба да бидат дефинирани во репозиториум од типот `JpaRepository`.

На пример, за да се дефинира барање кое ќе ги врати сите нарачки од класата `Order` на клиент со даден идентификатор `id` и статус `status` се користи следниот код:

```

1 interface OrderRepository extends JpaRepository<Order, Integer>{
2     @Query("select o from Order o where o.customer.id = :id and o.status =
3             :status ")
4     public List<Order> findByCustomerIdAndStatus(@Param("id") Integer
5             customerId, @Param("status") OrderStatus status);
6 }
```

Иако самото име на методот има валидна синтакса за преребарување, поради дефиницијата на анотацијата `@Query`, името воопшто нема да се евалуира, а за имплементација ќе се земе JPQL прашањето дефинирано во самата анотација. Во примерот, прашањето прима надворешни параметри кои се пренесуваат како аргументи на методот. За да се определи еднозначно на кој параметар се однесува секој од аргументите, се користи анотацијата `@Param` во која како аргумент се пренесува името на параметарот кое се користи во JPQL барањето. На пример, аргументот `customerId` се однесува на параметарот `:id`, па затоа тој се анотира со `@Param("id")`.

Постои и друг начин на означување на параметрите во барањето според редоследот по кој се појавуваат во листата на аргументи на методот. Во тој случај, не се користи анотација и секој параметар во барањето се означува со `?` и негови-

от реден број почнувајќи од 1. Според овој начин на пренесување на параметри, пребарувањето од претходниот пример би се дефинирало со кодот:

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

Друг начин на креирање корисничко дефинирано барање е преку користење на нативен SQL код. За таа цел, повторно се користи анотацијата `@Query` со аргумент `value` во кој се пренесува нативниот SQL код и дополнителен аргументот `nativeQuery=true`.

Барањето од претходниот пример преку нативен SQL код се дефинира на следниот начин:

```
1 @Query(value="select * from eshop_order o where o.customer_id = ?1 and
2     o.status = ?2 ", nativeQuery=true)
3 public List<Order> findByCustomerIdAndStatus(Integer customerId,
    OrderStatus status);
```

Сите кориснички дефинирани пребарувања можат да се користат со опцијата за подредување или страничење ако како последен аргумент на методот се пренесе инстанца од класата `Sort` или `Pageable`. Во случај на страничење, податоците кои ги враќа методот мора да бидат од типот `Page<T>`.

Ако сакаме во претходниот пример да имплементираме страничење, без разлика дали станува збор за нативно барање или не, дефиницијата на методот за пребарување треба да биде:

```
1 @Query("select o from Order o where o.customer.id = ?1 and o.status = ?2 ")
2 public Page<Order> findByCustomerIdAndStatus(Integer customerId,
    OrderStatus status, Pageable pageable);
3 }
```

Сите претходни барања се однесуваат на операцијата селекција (`select`). Ако се користат кориснички дефинирани барања кои прават модификација на податоците како што се бришење и уредување (`delete` и `update`), тогаш со самата дефиниција на барањето потребно е да се додаде и анотацијата `@Modifying`.

На пример, за да дефинираме метод кој ќе ја намали сумата на сите нарачки кои треба да ги плати клиентот со идентификатор `id` за одреден процент `discount` се користи следниот код:

```
1 @Modifying
2 @Query("update Order o set o.amount = (1-:discount*1.0/100)*o.amount where
```

```

    o.customer.id = :id ")
3 public List<Order> updateCustomerIdAmount(@Param("id") Integer customerId,
4     @Param("discount") Integer discount);
}

```

Spring Data JPA нуди можност и за други напредни функционалности кои се надвор од целите на книгата, како што се:

- Складирани процедури (анг. Stored Procedures)⁵ - процедури кои се напишани во нативен SQL код и се креирани во базата на податоци. Тие може да се повикуваат со или без аргументи, и можат да враќаат повеќе податоци одеднаш, вклучувајќи примитивни типови и цели табели од ставки.
- Спецификации (анг. Specification)⁶ - класа која овозможува динамичко граѓење на пребарување додавајќи предикати во пребарувањето во зависност од тоа кои параметри за пребарување се расположливи во повикот.
- Ревизија (анг. Auditing)⁷ - водење на дневник за модификација на ентитетите кој вклучува кориснички и временски податоци за модификацијата
- Именувани ентитетски графови (Named Entity Graphs) за подобрување на перформасните при читање на податоците

5.8.4 Вежба: Работа со Spring Data JPA

Во оваа вежба ќе го замениме постоечкиот репозиториум за CRUD и останати специфични операции врз ентитетите од класата `Address` со Spring Data JPA репозиториум и ќе го разгледаме автоматското креирање на барања преку методите кои се дел од `JpaRepository`, но пареку методи според конвенцијата за именување и кориснички дефинирани методи со помош на JPQL.

Во пакетот `repository`, креирајте репозиториум `AddressJpaRepository` како во изворниот код 5.33:

```

1 public interface AddressJpaRepository extends JpaRepository<Address,
2     Integer> {
}

```

Изворен код 5.36: Содржина на репозиториумот Spring Data JPA репозиториум `AddressJpaRepository` за работа со ентитетот `Address`

Променете ја содржината на апликациската класа така што ќе се декларира променлива од типот `AddressJpaRepository` и ќе се прошири конструкторот за

⁵<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.stored-procedures>

⁶<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#specifications>

⁷<https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#auditing>

автоматско да ѝ се додели вредност преку вметнување на зависности. Во рамките на класата, ќе го дефинираме методот `runAddressRepositoryExample()` кој ќе се повика преку бинот `commandLineRunner`. Делумната содржината на апликациската класа е дадена во изворниот код [5.37](#).

```

1 ...
2 final AddressJpaRepository addressSpringRepository;
3
4 public Ch4ExamplesApplication(AddressRepository addressRepository,
5                               CustomerRepository customerRepository, OrderRepository orderRepository,
6                               AddressJpaRepository addressSpringRepository) {
7     ...
8     this.addressSpringRepository = addressSpringRepository;
9 }
10
11 public static void main(String[] args) {
12     SpringApplication.run(Ch4ExamplesApplication.class, args);
13 }
14 @Bean
15 public CommandLineRunner commandLineRunner(ApplicationContext ctx ) {
16     return args -> {
17         runAddressRepositoryExample();
18     };
19 }
20 public void runAddressRepositoryExample(){
21     List<Address> addressList = new ArrayList<>();
22     addressList.add(new Address("Happiness Blvd.", 2, "Joy Town", 1000,
23                      "Dreamland"));
24     addressList.add(new Address("Happiness Blvd.", 3, "Joy Town", 1000,
25                      "Dreamland"));
26     addressList.add(new Address("Fun Av.", 22, "Party Town", 2000,
27                      "Dreamland"));
28     addressList.add(new Address("Funniest Av.", 52, "Party Town", 2500,
29                      "Dreamland"));
30     addressList.add(new Address("Mountain St.", 1, "Nature City", 1200,
31                      "Restland"));
32     addressList.add(new Address("Lake cross", 75, "Nature City", 1280,
33                      "Restland"));
34     addressList.add(new Address("Sea shore bay ", 10, "Nature City", 9040,
35                      "Restland"));
36     addressList.add(new Address("Hard work St.", 62, "Jobville", 6100,
37

```

```

    "Realand"));
28   addressList.add(new Address("Late home Sq.", 83, "Jobville",6120,
    "Realand"));
29   addressList.add(new Address("Late home Sq.", 80, "Jobville",6120,
    "Realand"));

30   System.out.println("----Spring Data JPA: saveAll----");
31   addressSpringRepository.saveAll(addressList);
32   Address address = addressList.get(0);
33   System.out.println("addressList[0]: "+address.toString());
34   address.setNumber(9);

35   System.out.println("----Spring Data JPA: save----");
36   addressSpringRepository.save(address);

37   System.out.println("----Spring Data JPA: findById----");
38   address = addressSpringRepository.findById(address.getId()).get();
39   System.out.println("addressList[0] modified: "+address.toString());

40   System.out.println("----Spring Data JPA: delete----");
41   addressSpringRepository.delete(address);
42   System.out.println("----Spring Data JPA: count----");
43   System.out.println("Total items: " + addressSpringRepository.count());

44   Pageable pageable = PageRequest.of(1,3);
45   System.out.println("----Spring Data JPA: findAll(pageable) no sort----");
46   Page<Address> page = addressSpringRepository.findAll(pageable);
47   System.out.println("page 1: "+page.toList());

48   System.out.println("----Spring Data JPA: findAll(pageable) with
      sort----");
49   pageable = PageRequest.of(0,3, Sort.by("number").descending());
50   page = addressSpringRepository.findAll(pageable);
51   System.out.println("page 0, sortBy number: "+page.toList());
52   System.out.println("Total elements: "+page.getTotalElements());
53 }

54 }

```

Изворен код 5.37: Содржина на апликациска класа за работа со ентитетот Address преку JpaRepository

Откако ќе се изврши апликацијата, најпрво се креира листа од адрески во која се сместуваат десет инстанци. Наместо да се перзистираат инстанците во базата

поединично, во примерот се користи методот `saveAll()` кој е дел од интерфејсот `JpaRepository` на Spring Data JPA кој зема цела листа на инстанци и ги зачува одеднаш во куп (анг. batch). Како резултат на оваа операцја, на конзолата се испишува следната содржина:

```
1 ----Spring Data JPA: saveAll----  
2 ... sequence generation ...  
3 Hibernate: insert into address (city, country, number, postal_code, street,  
     id) values (?, ?, ?, ?, ?, ?)  
4 ... insert for each instance  
5 adressList[0]: Address(id=1, street=Happiness Blvd., number=2, city=Joy  
     Town, postalCode=1000, country=Dreamland)
```

За секоја инстанца се генерира секвентен број кој потоа се користи за да се внесе редица во табелата `address` преку SQL наредбата `insert`.

Потоа, се зема првата инстанца од листата, се печати нејзината содржина, се менува вредноста на нејзионото свойство `number`, се перзистира новата состојба во базата преку методот `save` и се повлекува последната состојба на таа инстанца од базата преку `findById`.

Според испечатениот код на конзолата, содржината на инстанцата пред и по промената ќе биде:

```
1 addressList[0]:  
2     Address(id=1, street=Happiness Blvd., number=2, ...)  
3 addressList[0] modified:  
4     Address(id=1, street=Happiness Blvd., number=9, ...)
```

За да се прочита и зачува инстанцата, Spring Data JPA преку JPA провајдерот ги генерира `update` и `select` SQL барања на идентичен начин како и барањата во имплементациите каде експлицитно ги повикувавме методите од `EntityManager`, односно:

```
1 ----Spring Data JPA: save----  
2 Hibernate: select ... from address address0_ where address0_.id=?  
3 Hibernate: update address set city=?, country=?, number=?, postal_code=?,  
     street=? where id=?  
4 ----Spring Data JPA: findById----  
5 Hibernate: select ... from address address0_ where address0_.id=?
```

Во следните линии се брише инстанцата `address` која се однесува на првиот елемент од листата преку методот `delete()` и за да се види дека навистина е намален бројот на редици во базата се повикува методот `count()`. На конзолата се запишува:

```
1 Total items:9
```

Издадените SQL прашања за овие операции се `select` и `delete` за бришење и `count` за преbroјување на редиците во табелата `address`:

```
1 ----Spring Data JPA: delete----
2 Hibernate: select ... from address address0_ where address0_.id=?
3 Hibernate: delete from address where id=?
4 ----Spring Data JPA: count---
5 Hibernate: select count(*) as col_0_0_ from address address0_
```

На крајот, се извршуваат две варијанти на методот `findAll()` каде се преба-
рува само една страница од целата колекција на ставки и тоа со големина од
три ставки. Во првата варијанта се побарува втората страница (со индекс 1), а
во втората варијанта се побарува првата страница (со индекс 0) добиена на тој
начин што листата најпрво се подредува според вредноста на својството `number`.
За да се дефинираат индексот на страницата, нејзината големина, и опционал-
но, насоката на подредување и својството за подредување, се користи методот
`PageRequest.of()` кој враќа инстанца од `Pageable`. За подредување се користи
методот `Sort.by()` кој враќа инстанца од типот `Sort`. Иако не е прикажано во
примерите, `findAll()` може да прими како аргумент само инстанца од `Sort` кога
не е потребно страничење. Со оглед на тоа што `findAll()` со аргументи за страни-
чење враќа инстанца од класа `Page`, се користи нејзиниот метод `toList()` за да
се добие содржината на страницата во форма на листа. Во последната линија се
повикува и методот `getTotalElements()` од оваа класа за да се добие вкупниот
број на елементи без разлика за која страница станува збор.

Листите добиени од повикот на двета методи и вкупниот број на елементи
испечатени на конзолата ќе бидат:

```
1 page 1:
2     [Address(id=5, street=Mountain St., number=1, city=Nature City,
3             postalCode=1200, country=Restland),
4      Address(id=6, street=Lake cross, number=75, city=Nature City,
5              postalCode=1280, country=Restland),
6      Address(id=7, street=Sea shore bay , number=10, city=Nature City,
7              postalCode=9040, country=Restland)]
5 page 0, sortBy number, desc:
6     [Address(id=9, street=Late home Sq., number=83, city=Jobville,
7             postalCode=6120, country=Realand), Address(id=10, street=Late home
8                 Sq., number=80, city=Jobville, postalCode=6120, country=Realand),
9      Address(id=6, street=Lake cross, number=75, city=Nature City,
10             postalCode=1280, country=Restland)]
7 Total items: 9
```

Двата повика на методот `findAll()` со страничење ги генерираат следните SQL наредби:

```

1  ----Spring Data JPA: findAll(pageable) no sort----
2  Hibernate: select ... from address address0_ limit ?, ?
3  Hibernate: select count(address0_.id) as col_0_0_ from address address0_
4  ----Spring Data JPA: findAll(pageable) with sort----
5  Hibernate: select ... from address address0_ order by address0_.number desc
   limit ?
6  Hibernate: select count(address0_.id) as col_0_0_ from address address0_

```

Од прашањата може да се заклучи дека кога има страничење, секогаш генерира едно `select` прашање чиј број на резултати е ограничен со `limit` и едно `count` прашање за добивање на вкупниот број резултати без разлика дали е повикан методот `getTotalElements()`. Причината за вториот повик е што при страничењето, секогаш е потребно да се добие информација во колку страници се распоредени ставките за да може на корисникот да му се прикажат информации за избор на страници, што едноставно може да се добие со деление на бројот на елементи со големината на страницата. Ако дополнително се користи и подредување, се користи `order by` за дефинирање на колоната по која се подредува и `desc` (или `asc`) за дефинирање на насоката.

Сите досегашни примери сведочат дека не е потребно да се напише ниту една линија JPQL или SQL код за имплементација на методите кои се дел од дефиницијата на `JpaRepository`. Развивачот на софтвер треба само да ги повика, а за нивната имплементација е одговорен Spring Data JPA.

Наредно, ќе додадеме нови изведенни методти и кориснички дефинирани методти во репозиториумот `AddressJpaRepository` даден во изворниот код [5.38](#).

```

1  public interface AddressJpaRepository extends JpaRepository<Address,
   Integer> {
2
3      public List<Address> findByStreetIsEndingWithIgnoreCase(String end);
4
5      public List<Address> getByNumberBetweenAndPostalCodeGreaterThanOrEqual(
6          startNumber, Integer endNumber, Integer postalCode);
7
8      public Page<Address>
9          readByCountryNotInOrderByPostalCodeDesc(List<String> countryList,
10             Pageable pageable);
11
12     @Query("select a from Address a where a.city like ?1%")
13     public List<Address> myFindByCityContaining(String content);
14
15

```

```

9   @Modifying
10  @Transactional
11  @Query("update Address a set a.city = :city where
12    a.postalCode=:postalCode")
13  public void updateByPostalcode(@Param("postalCode") Integer postalCode,
14    @Param("city") String city);
}

```

Изворен код 5.38: Содржина на репозиториумот Spring Data JPA репозиториум AddressJpaRepository со изведени и кориснички дефинирани методи за работа со ентитетот Address

Потоа, во методот runAddressRepositoryExample() ги додаваме следните линии код за повикување на новите методи дефинирани во репозиториумот:

```

1 System.out.println("----Spring Data JPA: findByStreetEndingWith() ----");
2 addressList =
3   addressJpaRepository.findByStreetIsEndingWithIgnoreCase("St.");
4 System.out.println("Addresses with street ending with 'St.':");
5 System.out.println(addressList.toString());
6
6 System.out.println("----Spring Data JPA:
7   getByNumberBetweenAndPostalCodeGreaterThan(50,85,1200) ----");
7 addressList =
8   addressJpaRepository.getByNumberBetweenAndPostalCodeGreaterThan(50,85,1200);
9 System.out.println("Addresses with number between 50 and 85 and zip >
10   1200:");
9 System.out.println(addressList.toString());
10
11 System.out.println("----Spring Data JPA:
12   readByCountryNotInOrderByPostalCodeDesc(countryList, pageable) ----");
12 List<String> countryList = new ArrayList<>();
13 countryList.add("Dreamland");
14 countryList.add("Restland");
15 addressList =
16   addressJpaRepository.readByCountryNotInOrderByPostalCodeDesc(countryList,
17     PageRequest.of(0,2)).toList();
16 System.out.println("Addresses in first page not in list, ordered by zip
18   desc:");
17 System.out.println(addressList.toString());
18
19 System.out.println("----Spring Data JPA: updateByPostalcode(6120, Jobtown)
19   ----");

```

```

20 addressJpaRepository.updateByPostalcode(6120, "Workburg");
21 System.out.println("Addresses city containing 'Work':");
22 System.out.println(addressList.toString());
23
24 System.out.println("----Spring Data JPA: myFindByCityContaining(Town)
25      -----");
25 addressList = addressJpaRepository.myFindByCityContaining("Work");
26 System.out.println(addressList.toString());

```

По извршување на апликацијата на конзолата ќе се испечатати содржината на листите добиени како резултат на повиците на изведените и кориснички дефинирани методи од репозиторимот:

```

1 Addresses with street ending with 'St.':
2 [Address(id=5, street=Mountain St., number=1 ...),
3 Address(id=8, street=Hard work St., number=62, ...)]
4
5 Addresses with number between 60 and 80 and zip > 1200 ordered by number
   desc:::
6 [Address(id=10, ..., number=80, ..., postalCode=6120, ...),
7 Address(id=6, ..., number=75, ..., postalCode=1280, ...),
8 Address(id=8, ..., number=62, ..., postalCode=6100, ...)]
9
10 Addresses in first page not in list, ordered by number asc:
11 [Address(id=8, ..., number=62, ..., country=Realand),
12 Address(id=10, ..., number=80, ..., country=Realand),
13 Address(id=9, ..., number=83, ..., country=Realand)]
14
15 Addresses with city containing 'Work':
16 [Address(id=9, ..., city=Workburg, ...),
17 Address(id=10, ..., city=Workburg, ...)]

```

Пријот изведен метод ги пребарува сите адреси кои завршуваат на низа која се пренесува како аргумент, без разлика дали станува збор за мали или големи букви. За повиканиот метод `findByStreetIsEndingWithIgnoreCase("St.")` се добива листа од две адреси кои го исполнуваат условот.

Вториот метод ги пребарува сите адреси чии броеви се во даден опсег и кои имаат вредност на поштенскиот број поголема од зададената. Почетокот и крајот на опсегот како и граничната вредност на поштенскиот број се пренесуваат како аргументи на методот. Во името на методот дополнително се користи опција за специфицирање на својството според кое ќе се

врши подредување и редоследот на подредувањето. За повиканиот метод `getByNumberBetweenAndPostalCodeGreaterThanOrOrderByNumberDesc(60,80,1200)` се добива листа од три адреси подредени во опаѓачки редослед според својствоот `number`.

За да се повика третиот метод `readByCountryNotIn(countryList, Sort.by("number"))`, најпрво се креира листа од две земји `countryList` која ќе се користи како аргумент на методот според кој ќе се пребаруваат сите адреси чија адреса не припаѓа во таа листа. За разлика од претходниот метод, подредувањето се дефинира преку аргументот од типот `Sort` во кој се наведува колоната според која ќе се подредат резултатите. Бидејќи насоката не е наведена, како предефиниран се зема растечки редослед.

Следниот метод кој се повикува е кориснички дефиниран со помош на JPQL во анотацијата `@Query` и се користи за промена на податоци во базата на податоци. Затоа при неговата дефиниција се користат анотациите `@Modifying` и `@Transactional`. Параметрите во JPQL барањето се пренесуваат преку нивните имиња и за да се направи пресликување на имињата на параметрите во барањето со аргументите на методот се користи анотацијата `@Param` во која како аргумент се пренесува името на параметарот и се поставува пред аргументот во методот на кој се однесува тој параметар. Методот го менува својството `city` на сите адреси кои имаат одредена вредност на својството `postalCode`. Новата вредност на својството `city` и вредноста на `postalCode` се пренесуваат како аргументи. Така, повикот на `pdateCityByPostalcode(6120, "Workburg")` поставува вредност на `city="Workburg"` на сите адреси со `postalCode=6120`.

Последниот метод е исто така кориснички дефиниран со помош на JPQL и ги пребарува сите адреси кои содржат одредена низа во името на својството `city`. Низата за споредба се пренесува како аргумент на методот и преставува параметар кој се користи во JPQL барањето. За разлика од претходно, параметарот се користи како ординален, односно неговото пресликување со аргументот е постигнува преку наведување на редниот број на аргументот во листата на аргументи пред кој се прилепува знакот `?`. Со огледа на тоа што методот има само еден аргумент, тој се користи како параметар `?1` во JPQL барањето. Резултатот од извршување на `myFindByCityContaining("Work")` е листа од две адреси кои го исполнуваат условот.

SQL барањата кои се генерираат при повик на секој од разгледаните методи се дадени во продолжение:

¹ ----Spring Data JPA: `findByStreetIsEndingWithIgnoreCase()` -----

² Hibernate: `select ... from address address0_ where upper(address0_.street) like upper(?) escape ?`

```
3
4 ----Spring Data JPA:
      getByNumberBetweenAndPostalCodeGreaterThanOrEqualToNumberDesc() ----
5 Hibernate: select ... from address address0_ where (address0_.number
      between ? and ?) and address0_.postal_code>? order by address0_.number
      desc
6
7 ----Spring Data JPA: readByCountryNotInOrderByPostalCodeDesc() ----
8 Hibernate: select ... from address address0_ where address0_.country not in
      (? , ?) order by address0_.number asc
9
10 ----Spring Data JPA: updateCityByPostalcode() ----
11 Hibernate: update address set city=? where postal_code=?
12
13 ----Spring Data JPA: myFindByCityContaining(Town) ----
14 Hibernate: select ... from address address0_ where address0_.city like ?
```

Глава 6

Заштита на веб апликации

6.1 Што сè треба да се заштитува?

Веб апликацијата кој сме ја развиле и серверот на кој сме ја поставиле за да биде достапна преку Интернет мора да бидат заштитени. Защитата може да се направи на повеќе места, во различните слоеви на стекот на протоколи кои се користат за да се пристапи до неа, но и во самата апликација.

Ако станува збор за заштита на мрежниот слој, тогаш таа се сведува на ограничување во однос на IP пакетите кои се разменуваат помеѓу апликацијата и корисниците. Познато е дека овие пакети, покрај податоците од транспортниот слој, во заглавјето ги носат адресите на испраќачот и примачот. Затоа, во мрежниот слој може да се специфицираат кои адреси адреси се дозволени, а кои не се дозволени да пристапуваат до серверот или па ликацијата. Исто така, може да се постават и одредени филтри на пакетите според некое поле од заглавјето. Сепак, кај мрежниот слој опциите во однос на тоа што може да се заштите се прилично ограничени.

При заштита на транспортниот слој може да се постават ограничувања во однос на информациите кои ги носат заглавјата на сегментите¹ кои се разменуваат помеѓу клиентите и веб апликацијата. Некои од поважните полиња во заглавјето се портите кои преставуваат некој тип на адреса на апликацијата во рамките на компјутерскиот систем. За да се воспостави комуникација со различни сервиси на серверите, потребно е тие да слушаат на добро позната порта. На пример, веб серверот слуша на порта 80 и токму преку оваа порта прелистувачот воспоставува TCP конекција за размена на барања и одговори со серверот. Од аспект на конфигурација на серверот, треба да се внимава кои порти се отворени, односно до кои порти може да се пристапи од целиот Интернет. Постојат алатки

¹Сегменти се пораки кои што се разменуваат на транспортно ниво

како што е *nmap*², преку кои напаѓачот може да дознае кои порти на нашиот сервер се достапни. Ако некои од сервисите кои слушаат на соодветните порти имаат познати ранливости, тие може да се искористат од страна на напаѓачите и подоцна да се корумпира целиот сервер на кој е поставена апликацијата, а со самото тоа се корумпира и апликацијата која сме ја развиле. Најчести порти кои се отворени се портите 22 (порта преку која што може да се поврзeme со SSH на самиот сервер и да го конфигурираме)³, 80 (за HTTP протоколот) и 443 (за HTTPS протоколот). Сите останати порти треба да бидат затворени. Особено е важно да се оневозможи пристап до портите за комуникација со базите на податоци, како што е предефинираната порта 5432 за Postgres. Ако портата за базата на податоци е видлива од надвор и ако администраторот не ја променил предефинираната лозинка, тогаш напаѓачот може многу лесно да ги украде или избрише сите податоци во базата на податоци. Токму затоа, од голема важност е да се ограничи бројот на отворени порти и да се дозволи пристап само на оние порти кои се наменети да ги опслужуваат клиентите.

Постојат две опции за конфигурирање на дозволените порти:

- Да се конфигурира Firewall на ниво на инфраструктурата во која се наоѓа серверот, со кој ќе бидат достапни една до две порти на серверот.
- Да се конфигурира Firewall на ниво на оперативниот систем, каде повторно ќе се ограничат достапните порти.

Дополнително, на транспортниот слој се препорачува да се користи Secure Sockets Layer (SSL), односно податоците кои патуваат од и до серверот да се енкриптирани (секција 1.9). IP пакетите што поминуваат низ низа рутери може да бидат пресретнати и прочитани од некој напаѓач. Доколку не се користи SSL, напаѓачот може единствено да ја прочита содржината на целиот пакет, па дури може и да ја промени. Со SSL, доколку некој ги пресретне IP пакетите кои се разменуваат и ја извлече нивната содржина, нема да може да ја интерпретира без да ги поседува соодветните енкрипциски клучеви.

Заштитата можеме да се разгледува и на ниво на оперативниот систем. На пример, важно е да се внимава какви привилегии се дозволени на корисниците на апликацијата. Доколку е овозможена функционалност за прикачување на датотеки (upload), и доколку оваа функционалност запишува податоци во именикот */opt/app_name/uploads*, на тој именик треба да му се доделат соодветни привилегии во однос на тоа кој смее да запишува, кој смее да менува, кој смее да чита, и дали некој смее да извршува апликации, односно дали смее да има привилегија

²<https://nmap.org/>

³Иако оваа порта е често достапна на серверите, се препорачува таа да не е достапна преку Интернет, туку да користиме виртуелна приватна мрежа (VPN) за пристап преку SSH за конфигурација на серверот.

за извршување (*execute*) на самиот тој именик. Доколку во именикот *uploads* се ненамерно се остави привилегијата за извршување, напаѓачот може да прикачи некоја скрипта како датотека и подоцна може да ја повика таа скрипта, со што може да го корумпира целиот оперативен систем. Затоа, мора да сме особено внимателни со привилегиите на локациите каде што се зачувуваат прикачени-те датотеки со цел да ја избегнеме можноста за потенцијално корумпирање на целиот систем од страна на потенцијални напаѓачи. Препораката е да се даваат минимум привилегии за функционирањето на апликацијата.

Во случаите кога има повеќе апликации на ист сервер, треба да се внимава да бидат што е можно повеќе изолирани. Причината е поради тоа што ако една апликација има проблеми и е нападната, тоа не треба да влијае и врз другите апликации и да им предизвика проблем. Во ваквите ситуации, најчесто се користи виртуализација, односно поставување на различни апликации на различни виртуелни машини на еден физички сервер. Неретко се користи и контејнеризација (како Docker или нејзини алтернативи) со што делумно се штеди на ресурси во однос на виртуализацијата, но сепак се постигнува посакуваната изолација помеѓу различните апликации.

И покрај големиот број на начини на заштита, главниот фокус на ова поглавје ќе биде заштита во рамките на самата апликација која ја развиваме. Главното прашање кое ќе го обработиме е: Што можеме да направиме во рамките на апликацијата за да ги заштитиме нашите корисници и нивните податоци? **Сапо:** И што е она што ние треба да го заштитиме? - ова ми лични на истото што и претходното За да одговориме на ова прашање, ќе ги искористиме механизмите овозможени од модулот Spring Security, со главен акцент на процесите за автентификација и авторизација. Притоа, ќе ги заштитиме следните димензии:

- **Интегритет** - способноста да се осигураме дека информациите кои се разменуваат преку Интернет не се променети на кој било начин од неовластено лице.
- **Автентичност** - способноста да го идентификуваме идентитетот на лицето или субјектот кој пристапува на веб апликацијата.
- **Доверливост** - способноста да се осигураме дека пораките и податоците се достапни само за оние кои се овластени да ги гледаат.
- **Приватност** - способноста да се контролира употребата на личните информации.
- **Достапност** - способноста да се контролира кога може да се пристапи апликацијата.

6.2 Напади на веб апликации

Во ова поглавје ќе ги обработиме некои од најчестите напади на кои треба да бидеме подготвени и со кои треба да се справимвме, фокусирајќи се само на нападите кои се однесуваат на несоодветен дизајн на апликацијата, затоа што целта на оваа книга е развој на веб апликации.

6.2.1 Вметнување на SQL

Преку SQL вметнување се нарушуваат интегритетот и приватноста на податоците во рамките на апликацијата. Овој напад врши модификација на комуникацијата со базата на податоци, при што може да се променат податоците во неа, или пак да се добијат приватни информации кои не се дозволени за пристап за напаѓачот.

Пристапот кон база на податоци на најниско ниво се прави преку SQL прашање. Покрај прашања за селекција на податоци, постојат и типови на прашања за бришење, внесување и уредување на податоци. Ако не постои заштитата од вметнување на SQL, напаѓачот може лесно да внесе текст во поле од типот `input` на некоја нејзина страницата во веб прелисувачот со кое би ја избришал целата база или би ги украд сите податоци. Нападот со вметнување на SQL користи `input` свойство кое е наменето за внес на текст од страна на корисникот и кој најчесто се запишува во база на податоци. Напаѓачите ги користат овие свойства за да го корумпираат SQL прашањето кое го се испраќа кон базата на податоци.

На пример, ако имаме `login` форма со поле `username` за внес на корисничко име и ако во него ја внесеме вредноста `intruder';delete from users;`, постои ризик да се избришат сите корисници од базата на податоци. Проблемот е ако прашањето кое го извршуваме кон базата е од следниот формат: `"select * from users where username='"+username + "' and password='"+password + "'";`. Во овој случај, прашањето кое ќе се изврши кон базата на податоци ќе биде: `"select * from users where username='intruder'; delete from users; and password='';"`. Овде всушност имаме 3 прашања: првото прави селекција на корисник, второто ги брише сите корисници од базата и третото е невалидно.

Заштитата од вакви напади може да се постигне избегнеме спојување на текстуални податоци добиени од кориснички внесени податоци директно во прашањето кое го извршуваме кон базата на податоци. Решението е да ги избегнеме специјалните SQL карактери во рамките на влезовите кои ги добиваме од клиентот, со што не би се корумпирално прашањето кое се извршува кон базата на податоци. Ако решиме самите да пишуваме SQL прашања преку Java Database

Connection - JDBC, треба да бидеме внимателни барем да ги валидирате податоците кои се внесени од корисникот и да се осигураме дека на сите наводници ќе им се направи избегнување (escape), односно секој наводник ќе се замени со коса прта и наводник `"`). На сличен начин треба да ја избегнеме точка запирка (`;`) и сите останати валидни специјални знаци и синтакски елементи на SQL јазикот кои можат да го променат однесувањето на самото SQL прашање кое се испраќа до базата.

Сепак, доколку користиме веќе докажани библиотеки за пристап кон базите, можеме да добиеме заштита од овој тип на напад. На пример, библиотеката за JPA имплементација Hibernate врши избегнување на специјалните карактери на параметрите кои се испраќаат на прашањата, со што имплицитно прави заштита од овој тип на напади.

6.2.2 Cross-site scripting

Со нападот cross-site scripting се нарушуваат интегритетот и приватноста на податоците. Нападот вметнува извршен код како податок, кој доколку се прикаже на друг корисник, може неовластено да пристапи до неговите податоци и да ги украде (нарушување на приватноста) или да ги промени (нарушување на интегритетот).

Овој напад ќе го објасниме преку пример во апликација за пишување на вести од различни корисници. Доколку некој напаѓач напише извршен код (најчесто во javascript) и го зачува како вест, тогаш овој код ќе се изврши кај секој од клиентите кои ќе пристапат на соодветната вест. Ваквите кодови може да се искористат за крадење на дел од податоците на корисниците или злоупотреба на нивните ресурси за недозволени цели.

За да се заштитиме од cross-site scripting нападот, најдобро е да користиме докажани библиотеки кои при испртување на податоците прават избегнување на скриптите преку избегнување на присутните html елементи кои не се дозволени (output sanitation).

6.2.3 Фалсификување барања меѓу страници (Cross-Site Request Forgery – CSRF)

Со овој напад се нарушува автентикацијата и доверливоста на податоците, така што напаѓачот ги користи податоците кои се локално зачувани кај жртвата за да изврши акции без негово знаење. На овој начин, напаѓачот со помош на туѓи информации за автентикација и авторизација врши акции со кои дополнително се нарушува и приватноста и интегритетот на податоците.

Фалсификување барања меѓу страници (CSRF) е напад што ги принудува автентицираните корисници да поднесат барање до веб-апликација за која моментално се автентицирани. Нападот ја експлоатира ранливоста во веб-апликацијата ако таа не може да направи разлика помеѓу барање генерирано од индивидуален корисник и барањето генерирано од корисник без негова согласност.

На пример, ако имаме банкарска апликација, напаѓачот може да симулира барање до формата за плаќање и да побара на него лично да му се префрлат средствата. Напаѓачот најчесто користи анализа на формата за плаќање и ако имаме зачувано лични податоци во колаче на прелистувачот, може да се вметне невидлив код во рамки на некоја друга страница, преку кој ќе се испрати барањето за трансфер на средствата без знаење на корисникот. За ова барање да биде успешно, мора корисникот претходно да е најавен на банкарската апликација.

Validate: Напаѓачот треба да го натера корисникот да отвори веб страница со злонамерен код (на пример, ако кликнене на линк во меил со содржина од типот „добивте бесплатен телефон, кликнете тука за да ја подигнете наградата“), кој ќе ги украде неговите зачувани автентикациски колачиња од претходната сесија и ќе генерира барање во негово име кон банкарската апликација за пренос на средства од сметката на корисникот на напаѓачот

За да се заштитиме од овој напад, можеме во сите форми да генерираме случаен токен кој за секоја наредна најава или за секое барање ќе биде различен. Со ова ќе бидеме сигурни дека некој нема да може да ни направи успешна анализа на барањата што ги практикаме, бидејќи нема да може да ја предвиди вредноста на овој CSRF токен и со самото тоа барањата до серверот ќе бидат невалидни. Со користење на Thymeleaf и овозможување на CSRF филтерот од Spring Security модулот, секогаш кога се рендерира страна, овој токен се вметнува во самите форми. Подоцна, кога корисниците ќе направат барање, на серверска страна се валидира дали тој токен е присутен и валиден. На овој начин се осигуруваме дека барањето е од вистински корисник, а не од напаѓач.

6.3 Основи на безбедност на системи

6.3.1 Терминологија за безбедност

При описување на концептите за безбедност и заштита на веб апликации ќе користиме специфична терминологија, па затоа е потребно истата да ја дефинираме. Најчесто користените клучни поиме од безбедноста се следните:

- **Корисник (User)** – концепт преку кој ги идентификуваме легитимните агенти на кои треба да им дозволиме пристап до системот. Најчесто корис-

ниците се дел од модел слојот на апликацијата и ги содржат неопходните информации преку кои може да ги идентификуваме во рамките на апликацијата.

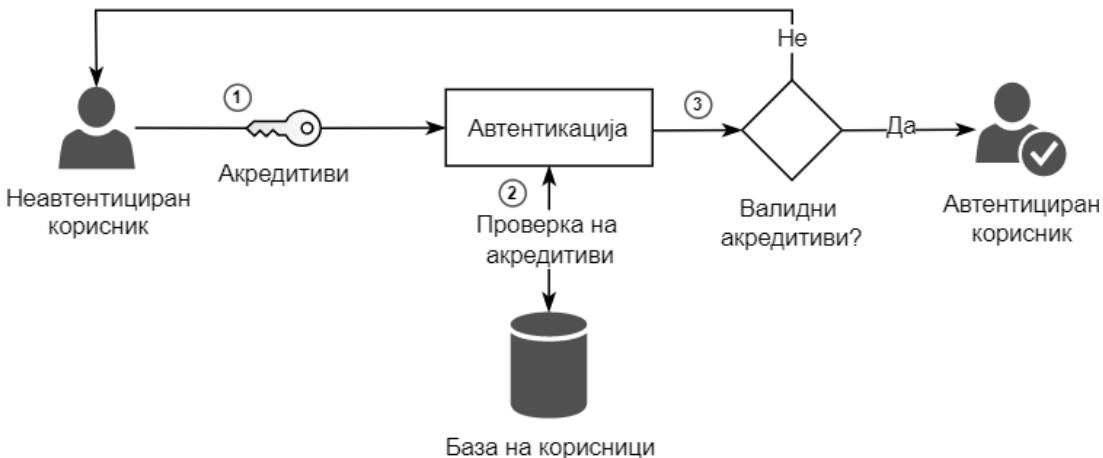
- **Акредитиви (Credentials)** – начинот на кој корисникот може да докаже дека тој е оној кој што вели дека е. Пример за акредитиви се лозинка (password), X509 сертификати и податоци од биометриски сензори. Ова се податоци кои единствено ги знае или поседува корисникот и со кои системот може да го потврди неговиот идентитет.
- **Улога (Role)** – групирање на корисници со цел да се олесни процесот на доделување на дозволи за пристап до ресурсите. Најчесто апликациите имаат огромен број на корисници, па затоа процесот на конфигурација на корисници е тежок и долготраен, особено ако се врши конфигурација за секој корисник посебно. Токму затоа, корисниците често ги групират според нивните улоги во бизнисот за кој се однесува апликацијата, па наместо на корисниците, дозволите ги доделуваме на улогите. Така, сите корисници кои се дел од една улога ги имаат истите дозволи кои ѝ се доделени на таа улога.
- **Ресурси (Resource)** – се она што сакаме да го заштитиме. Ресурсите може да бидат страници, патеки од програмскиот интерфејс, методи од бизнис логиката, класи од моделот, нивните својства и многу други елементи од апликацијата. Всушност, сите делови од апликациите за кои сакаме да го ограничиме пристапот за одредени корисници се третираат како ресурси.
- **Дозволи (Permissions)** – дефинираат кои ресурси се дозволени или забранети за одредени улоги или корисници.

6.3.2 Процеси во безбедност

Пред да продолжиме со конкретна имплементација на заштита на апликациите, најрво ќе ги дефинираме клучните процеси на поопшто ниво. **Сапо: уште некоја општа реченица**

Автентикација (Authentication)

Во процесот на автентикација, корисникот кажува кој е, односно му се претставува на системот. Многу често, овој процес се нарекува процес на најава (login process). Автентикацијата е неопходна за да знаеме кој пристапува на системот, а со тоа директно да ја адресираме димензијата за автентичност. Дополнително, без автентикација, не може да го авторизираме корисникот, а со тоа и индиректно да ги адресираме димензиите за доверливост и приватност.

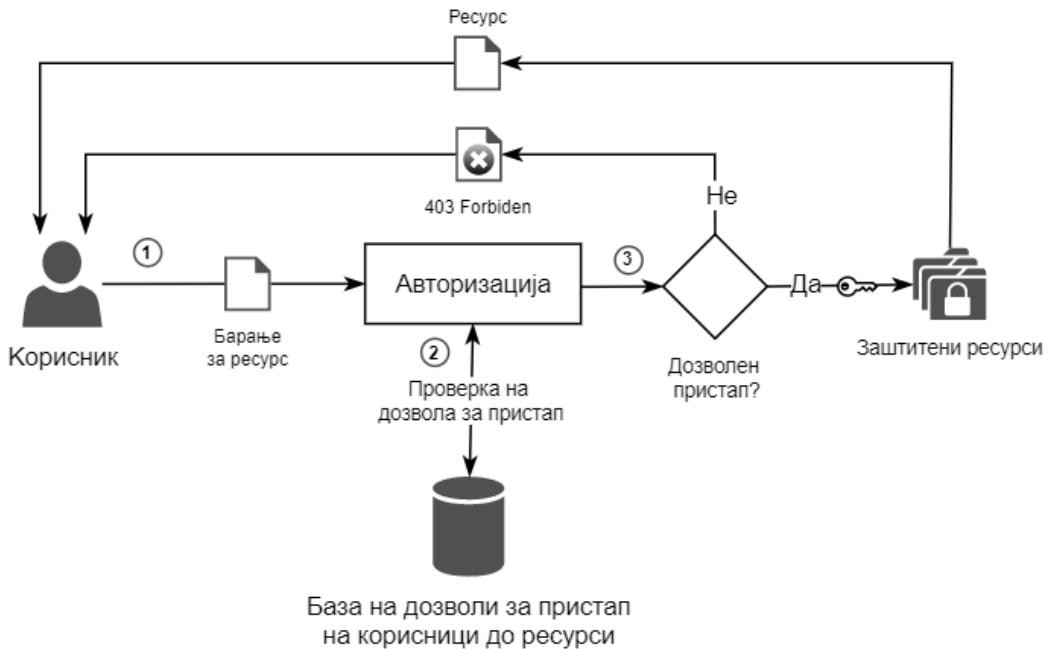


Слика 6-1: Процес на автентикација

Како што е прикажано на сликата 6-1, овој процес се иницира од корисник. Корисникот не секогаш мора да е човек. Корисник е апстракција за секој кој може да има интеракција со системот. За да знаеме за кој корисник се работи, потребно е да го идентификуваме. За оваа цел, корисникот треба дополнително да ги испрати своите акредитиви (credentials). Најчест акредитив се корисничко име и лозинка, но постојат и други форми кои може да се користат, како што се JSON Web Token (JWT) или X509 сертификати. Акредитивите се концепт со кој можеме да докажеме дека корисникот е тој кој што вели дека е. Овие акредитиви најчесто комбинираат информации кои само корисникот ги знае или поседува и кои што можеме да ги провериме. Кога акредитивите ќе стигнат до системот за автентикација, нивната валидност се проверува во базата на корисници. Ако се валидни, тогаш знаеме дека корисникот е тој што вели дека е. Во спротивно, корисникот не го автентицираме и најчесто бараме повторно да внесе валидни акредитиви, односно го пренасочуваме до страната за најава, каде треба да ги коригира акредитивите и повторно да ги испрати. Во случаите кога акредитивите не се валидни, најчесто се враќа статусен код 401 (Unauthorized) во одговорот кон клиентот.

Авторизација (Authorization)

Авторизацијата е задолжителен процес кој не смее никогаш да се изостави кога е во прашање безбедност. Преку авторизацијата ја штитиме димензијата за доверливост на апликацијата каде контролираме кој корисник какви интеракции има со ресурсите. На пример, во една апликација за човечки ресурси на компанија, секој вработен може да пристапи само до податоците за својата плата, но менаџерите може да ја видат целата табела за сите вработени и нивните пла-



Слика 6-2: Процес на авторизација

ти. Дополнително, овој процес се грижи и за заптита на приватноста каде се контролира пристапот до сензитивни приватни поадтоци.

Најчесто процесот на авторизација следи по комплетирањето на процесот на автентификација. Во одредени случаи можеме да имаме и авторизација на корисници кои не се автентицирани, но ваквите корисници ги третираме како да припаѓаат на посебна група (или улога) на анонимни корисници за која се доделуваат специфични дозволи за одредени ресурси. Пример за ваква дозвола е пристапот до страната за најава само за корисници кои не се автентицирани.

Во процесот на авторизација се прави проверка дали корисникот има дозвола за да има интеракција со бараниот ресурс. Во слојот на авторизација се проверуваат дозволите кои се конфигурирани за апликацијата. На слика 6-2 е прикажано дека се проверува *базата со дозволи за интеракција на корисниците до ресурсите*. Ова е всушност апстракција на начинот на чување на дозволите, кои најчесто се конфигурирани и се чуваат во меморија.

Кога станува збор за ресурси, често се конфигурираат дозволи за тоа кој смее да прави измени, кој смее да пристапи до ресурсот, кој смее да го избрише итн. Интеракцијата е всушност акцијата која се извршува врз ресурсот. Системот за авторизација прави проверка на дозволите кои веќе се конфигурирани или имплементирани во апликацијата за конкретниот корисник и за конкретниот ресурс. Доколку има дозвола за корисникот, тогаш велиме дека ресурсот е заштитен и дека корисникот може да продолжи понатаму. Доколку нема дозвола за корисни-

кот, тогаш се спречува интеракцијата со бараните ресурси. Кога интеракцијата не е дозволена, најчесто се генерира одговор со статусен код 403 (Forbidden).

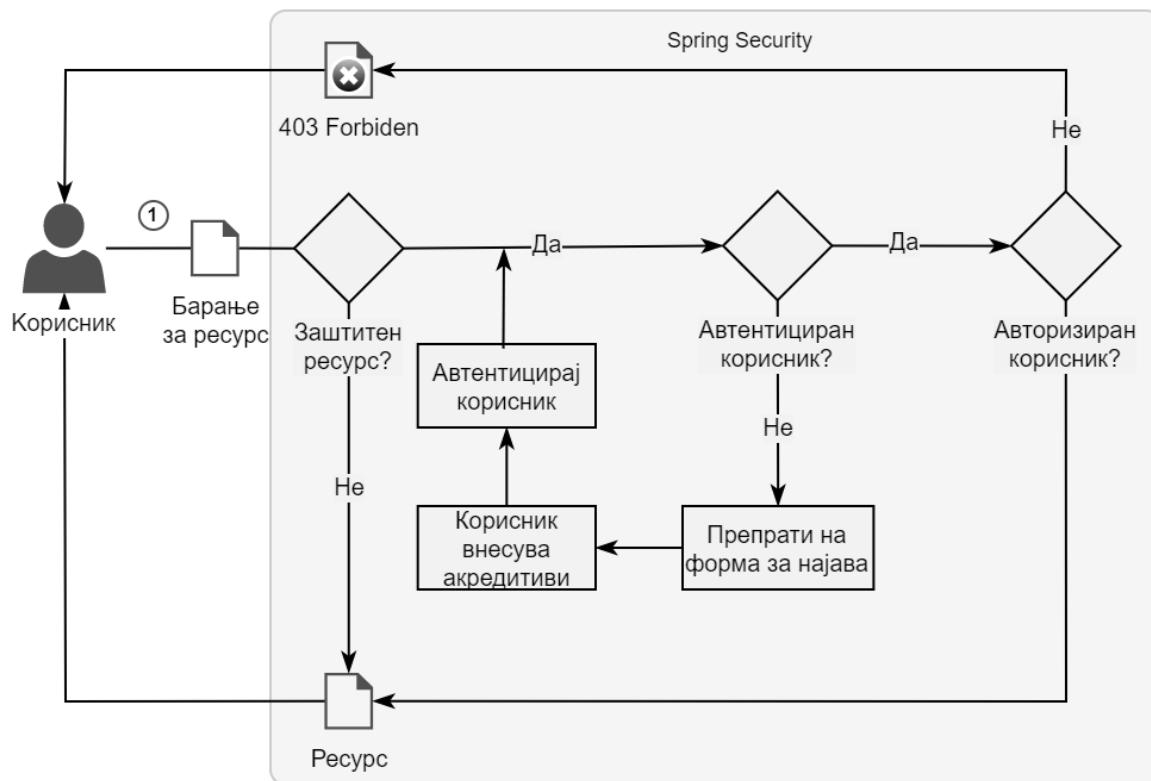
Енкрипција (encryption)

Енкрипција е процес преку кој сензитивните информации ги правиме неразбираливи за напаѓачите. До сега разгледавме дека еден начин да се енкриптираат податоците кои се разменуваат помеѓу прелистувачот и веб серверот на кој се хостира апликацијата е со користење на Transport Layer Security (имплементација на SSL протокол) протоколот на транспортно ниво, односно апликацискиот протокол HTTPS (секција 1.9).

Другото ниво на енкрипција, кое многу често се прави во пракса, е енкрипција на податоците во базата на податоци. Препорачливо е да се енкриптираат сензитивните податоци за да ги заштитиме од напад како протекување на податоци (анг. data breach). Протекување на податоци може да настане и при пропуст како што е заборавена отворена порта на која опслужува сервоерот со базата на податоци и непроменети предефинирани лозинки. Протекувањето на податоци не е секогаш поврзано со надворешен напад. Тоа може да настане и при внатрешен напад кога некој од вработените ќе пристапи до сензитивните податоци. Најчести податоци кои се енкриптираат во рамки на апликациите се лозинките. За оваа намена се користат алгоритми за еднонасочна енкрипција (хеширање), преку кои може само да се валидира точност на испратена лозинка наспроти енкриптираниот податок, но не може да се добие лозинката од информациите зачувани во системот. Со ваквиот пристап, кога корисникот се регистрира во системот, пред да се внесе во базата, лозинката се хашира, а потоа, при процесот на најава се прави хаширање на испратената лозинка и добиената вредност се сподредува со зачуваната хаширана лозинка во базата.

6.4 Заштита на веб апликации со Spring Security

Заштитата на веб апликации во работната рамка Spring е овозможена преку модулот Spring Security. Станува збор за релативно сложен модул кој има за цел да му олесни на развивајачот комплетно да ја имплементира безбедноста во апликацијата. Овој модул може да се замисли како дополнителен слој изграден над апликација кој ги обвитеа специфичните влезни точки во бизнис логиката со одредени безбедносни правила. Spring Security подржува конфигурабилни правила за авторизација на корисниците, заштита од некои од начестите напади, како и едноставни конфигурации за начинот на автентикација.



Слика 6-3: Преглед на заштита со Spring Security

За да се овозможи модулот Spring Security, треба да се додаде неговата зависност во *rom.xml*, како што е прикажано во изворниот код 6.1.

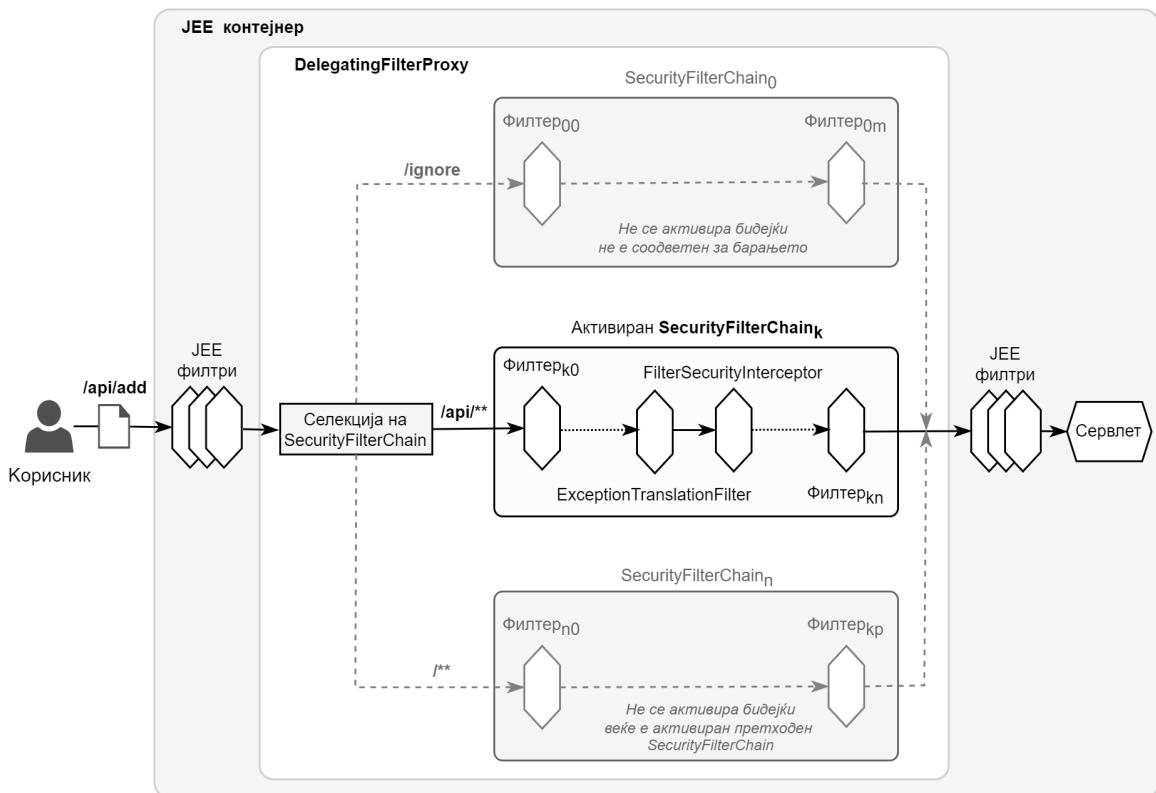
```

1 <dependencies>
2   <!-- ... -->
3   <dependency>
4     <groupId>org.springframework.boot</groupId>
5     <artifactId>spring-boot-starter-security</artifactId>
6   </dependency>
7 </dependencies>

```

Изворен код 6.1: Вклучување на Spring Security модулот во rom.xml

Најопшто гледано, користењето на веб апликацијата се сведува на испраќање барање од страна на корисникот и негово опслужување од страна на апликацијата. Кога се користи Spring Security во рамки на апликацијата (слика 6-3), тоа барање најпрво го пресретнува Spring Security, кој проверува дали ресурсот е заштитен. Ако ресурст не е заштитен, веднаш му се доставува на корисникот. Ако пак ресурст е заштитен, најпрво се порверува дали корисникот е автентициран. Неавтентициранот корисник се редиректира на форма за најава на која ги внесува акредитивите. Ако корисникот успешно се автентицира, се проверува



Слика 6-4: Повеќе синцири со филтри

дали има дозвола за пристап до конкретниот ресурс според конфигурацијата. Ако има дозвола, го добива ресурсот. Ако корисникот нема дозвола за пристап, добива порака за грешка со статусен код 403 Forbidden.

Имплементацијата на заштитата преку Spring Security се базира на користење на JEE филтри. На сликата 6-4, е прикажан процесот на безбедносна обработка на HTTP барањето пред да му се додели на сервлетот за негова обработка, во случај да му е дозволен пристапот. Барањето најрво го пресретнува `org.springframework.web.filter.DelegatingFilterProxy`, кој е стандарден JEE филтер дефиниран во рамките на Spring Security. Овој филтер повикува синцир од сигурносни филтри (Security Filter Chain) во зависност од патеката на побараниот ресурс. Синцирот со сигурносни филтри е имплементација на интерфејсот `org.springframework.security.web.SecurityFilterChain`. Неговите имплементации треба да кажат за кои барања треба да се повика синцирот со сигурносни филтри и кои филтри ги содржи овој синцир. Секој синцир со сигурносните филтри во себе содржи низа од конфигурирани имплементации на `javax.servlet.Filter`. Овие филтри **не се регистрирани** во JEE контејнерот како стандардни филтри, туку се управуваат од Spring Security и се вклучуваат преку конфигурацијата на синцирот со сигурносни филтри. Иници-

јалните филтри се вклучуваат во рамките на апликацијата преку анотацијата `@EnableWebSecurity`, која е композитна анотација што ги вклучува конфигурациите `WebSecurityConfiguration`, `SpringWebMvcImportSelector`, `OAuth2ImportSelector` и `HttpSecurityConfiguration`. Поконкретно, `HttpSecurityConfiguration` ги конфигурира филтрите кои ќе се регистрираат во предефинираниот синџир со сигурносни филтри.

`DelegatingFilterProxy` содржи референци до повеќе синџири со сигурносни филтри. Притоа, за секое барање проверува кој синџир со сигурносни филтри му одговара според конфигурацијата во апликацијата и за избраниот синџир редоследно ги повикува неговите филтри. Сигурносните филтрите се конфигурабилни и може да се прилагодат или оневозможат според потребите на апликацијата. На пример, за конфигурација на нов синџир со филтри `SecurityFilterChain` кој ќе се активира при барање на ресурси со патека `/api/**`, потребно е само да се вклучи инстанца од `SecurityFilterChain` во апликацискиот контекст на Spring апликацијата, како што е прикажано во изворниот код 6.2.

```

1  @Bean
2  public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
3      http.antMatcher("/api/**")
4          // the rest of the configuration
5      return http.build();
6  }

```

Изворен код 6.2: Код за вклучување на синџир на филтри кој ќе се повика при барање на ресурси со патека `"/api/**"`

Изворниот код 6.2 прикажува само како да се вклучи синџир од филтри, но за да се специфицираат конкретните сигурносни филтри кои ќе го обработуваат барањето заедно со нивните карактеристики се користи интерфејсот `HttpSecurity` кој се пренесува како аргумент на методот за регистрација. Филтрите кои се дефинирани како дел од Spring Security модулот може да се класифицираат во неколку групи:

- **Помошни:** се грижат за вчитување или зачувување на податоците кои се потребни во процесите на автентификација или авторизација. Примери за функции кои ги извршуваат овие филтри се вчитување и запишување на објектите кои го чуваат авторизираниот корисник, пренасочување на страницата за најава на неавтентицираните корисници кои сакаат да пристапат до заштитен ресурс, како и филтрите за справување со грешки.
- **Филтри за пресретнување на напади:** се грижат за превенирање на нападите од типот Cross Site Request Forgery и Cross Origin Request Sourcing и

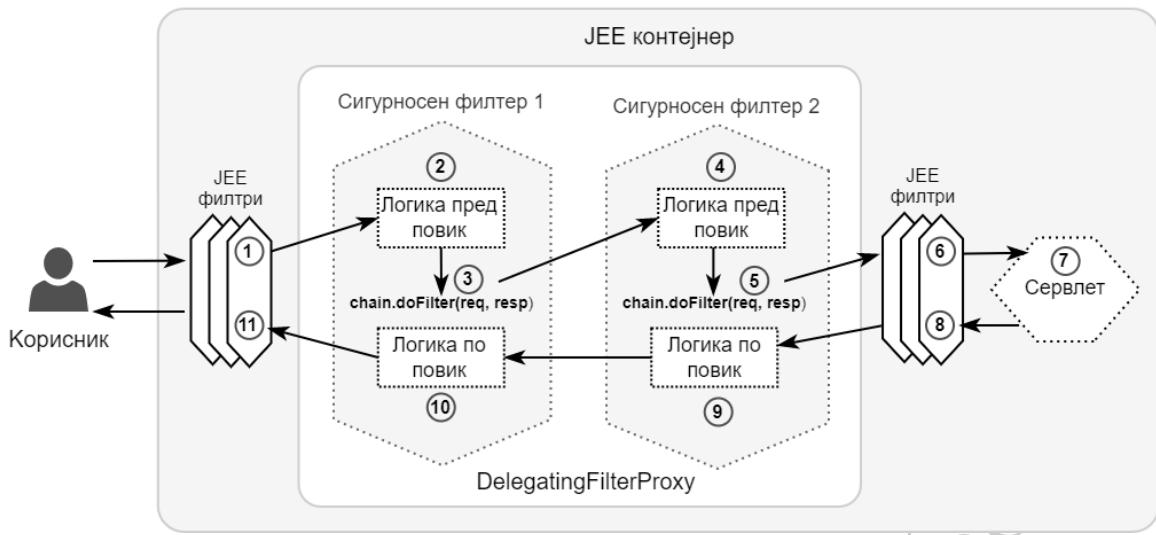
за зголемување на сигурноста на апликацијата со додавање на сигурностни заглавја.

- **Автентикациски:** се грижат за извршување на процесот на автентикација. За секој различен тип на автентикација, може да бидат регистрирани еден или повеќе филтри. Во оваа категорија припаѓаат и филтрите кои се задолжени за прикажување на страната за најава, како и филтрите за одјава на автентицираниот корисник.
- **Авторизациски:** се гришат за доделување на пристап до ресурсите. Претставник на оваа група е `FilterSecurityInterceptor` филтерот, кој проверува дали на корисникот му е дозволне пристап до бараниот ресурс. `FilterSecurityInterceptor` ја користи поставената конфигурација за заштита на тековното барање во однос на тековниот автентициран корисник за да одлучи дали корисникот има дозвола за да пристап до бараниот ресурс. Автентикацијата ја прават сигурносните филтри за оваа намена кои се извршуваат пред `FilterSecurityInterceptor` филтерот. Во верзиите по 5.7 на Spring Security модулот, овој филтер се напушта (се означува како `@Deprecated`) и се заменува со `AuthorizationFilter`. Во рамките на оваа книга и ќе ги поистоветиме `FilterSecurityInterceptor` и `AuthorizationFilter`, бидејќи извршуваат иста функција. Нивните разлики се во начинот на процесирање на дозволите, но поради комплексноста, тие нема да бидат разгледани во книгата.

Откако ќе биде избран синџирот на филтри, барањето се доделува на првиот сигурносен филтер кој извршува одредена операција, по што продолжува наредниот филтер во синџирот.

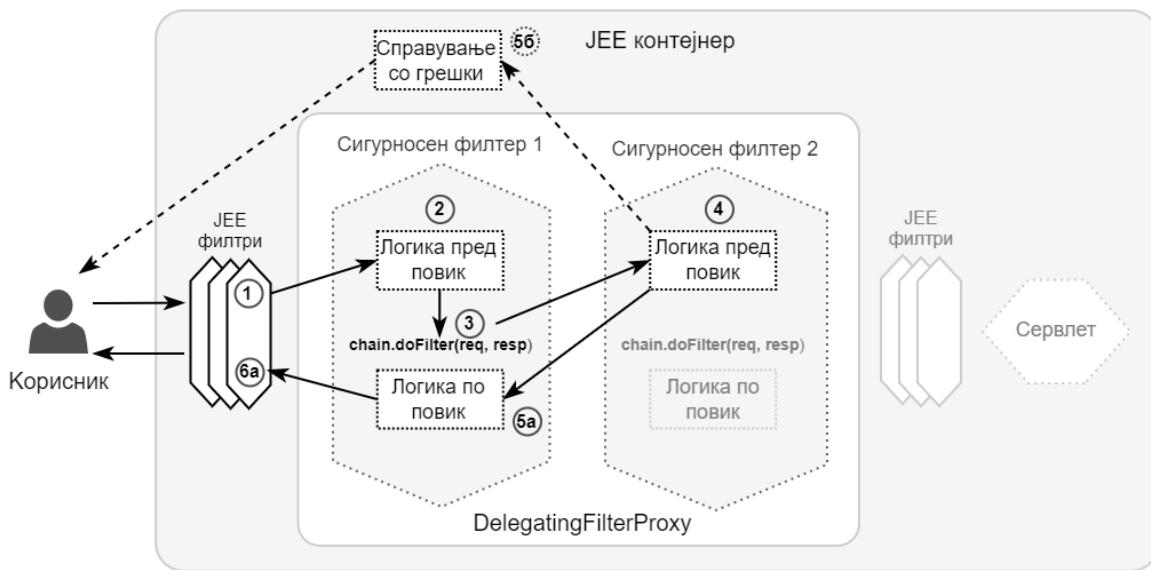
Пример за комплетна обработка на барање испратено од корисник со нормален тек на извршување на сите компоненти, вклучувајќи и безбедносен синџир со два филтри е прикажан на слика 6-5. Како што е веќе познато за JEE филтрите, филтерот кој обработува едно барање го повикува следниот филтер од синџирот преку методот `chain.doFilter(request, response)` (секција 2.5.2) и затоа, секој од филтрите има код кој се извршува пред да ја делегира контролата со повик на методот и дел кој се извршува откако сите наредни филтри завршиле и го проследуваат барањето назад во насока кон првиот повикан филтер. На сликата, овие два дела се означени како „Логика пред повик“ и „Логика по повик“. Откако барањето ќе стигне до сервлет контејнерот тоа се обработува на следниот начин:

1. Се повикуваат конфигурираните JEE филтри (не се од интерес во оваа глава). Секој од овие филтри го предава барањето на следниот филтер со повик на `doFilter()`.



Слика 6-5: Повикување на сигурносните филтри во нормално сценарио

2. Барањето се предава на DelegatingFilterProxy кој го одбира сигруносниот синџир на филтри кој одговара за патеката на барањето според конфигурацијата и го предава на првиот сигурносен филтер од синџирот кој ја извршува *логиката пред повик* на doFilter() методот.
3. Првиот филтер го повикува doFilter() и со тоа барањето го предава на вториот филтер од безбедносниот синџир на филтри.
4. Вториот сигурносен филтер ја извршува *логиката пред повик* на doFilter().
5. Вториот филтер го повикува doFilter() по што DelegatingFilterProxy го предава барањето на контејнерот.
6. Контејнерот го проседува барањето до остатокот од JEE филтрите кои се конфигурирани. Секој од овие филтри го предава барањето на следниот филтер со повик на doFilter().
7. Контејнерот го предава барањето на соодветниот сервлет кој генерира одговор.
8. Контејнерот го испраќа одговорот на последниот JEE филтрер. Секој од филтрите ја извршува логиката после повик на doFilter().
9. Вториот сигурносен филтер ја извршува *логиката после повик* на doFilter().
10. Првиот сигурносен филтер ја извршува *логиката после повик* на doFilter().
11. Одговорот се проследува на JEE филтрите кои се извршиле пред DelegatingFilterProxy, во насока од последниот кон првиот филтер сè додека првиот филтер не го предаде на контејнерот кој пак го испраќа одговорот.



Слика 6-6: Повикување на сигурносните филтри при исклучок или неисполнување на безбедносни услови

на клиентот.

При обработка на барањето секој од сигурносните филтри може да одлучи *да не ја делегира контролата* на наредните филтри, со тоа што под одредени услови нема да го повика `chain.doFilter(request, response)`. Сценариот каде вториот сигурносен филтер од претходниот пример не ја делегира контролата на наредните филтри е прикажано на слика 6-6. Оваа одлука се носи во логиката пред повикот на `doFilter` (чекор 4), па со оглед на тоа што методот не се повикува, барањето нема да се проследи ниту до останатите JEE филтри ниту до сервлетот (следствно, ниту до неговиот контролер). Во ваквите случаи, резултатот што ќе се врати на корисникот преку одговорот треба да се генерира во самиот филтер кој го спречил проследувањето на барањето поради нездовољување на потребните критериуми (чекор 4). Ваквиот одговор се проследува на сигурносниот филтер 1, кој откако ќе ја изврши логиката по `doFilter()` (чекор 5a) го проследува барањето кон останатите JEE филтри во насока кон првиот филтер кој го обработил барањето (чекор 6a).

Друго можно сценарио во процесот на обработката на барањето од сигурносните филтри е филтерот да фрли и исклучок и со тоа не само да го спречи извршувањето на останатите филтри и контролерот, туку и да го спречи и извршувањето на *локиката по повик* на `doFilter()` на филтрите кои се регистрирани пред него и кои му ја делегирале контролата како што е прикажано на слика 6-6. На слика, сигурносниот филтер 2 при извршување на логиката пред `doFilter` фрла исклучок по што исклучокот се обработува од страна на кодот за справување со

грешки (чекор 5б) кој е дел од JEE контејнерот. Откако грешката ќе се обработи, одговорот се враќа на корисникот. Кодот за справување со грешки може да биде имплементиран и во сигурносните фитри кои се извршуваат претходно, со што може да се обезбеди посоодветно справување со грешката.

6.4.1 Процес на автентикација и авторизација кај Spring Security

За подетално објаснување на имплементацијата на процесите за автентикација и авторизација кај Spring Security, ќе се користи пример во кој ненајавен корисник ќе се обиде да пристапи на патеката `"/admin/products"` за прв пат. Притоа, страната која одговара за оваа патека е заштитен ресурс, до кој пристап имаат само корисници со администраторска улога.

Конфигурација

За подетално објаснување на имплементацијата на процесите за автентикација и авторизација, ќе се користи конфигурацијата на синџирот со фитри во изворниот код [6.3](#). Во овој код, се дефинираат два бинови од типовите `WebSecurityCustomizer` и `SecurityFilterChain`. `WebSecurityCustomizer` бинот одредува за кои патеки воопшто да не се врши заштита, а во конкретниот случај тоа се патеките кои започнуваат со `/assets/`.

```

1 @Configuration
2 public class WebSecurityConfig {
3
4     @Bean
5     public WebSecurityCustomizer ignoringCustomizer() {
6         return (web) -> web.ignoring().antMatchers("/assets/**");
7     }
8
9     @Bean
10    public SecurityFilterChain configure(HttpSecurity http) throws Exception {
11        http.antMatcher("/**")
12            .authorizeRequests()
13            .antMatchers("/", "/home", "/register",
14                  "/products").permitAll()
15            .antMatchers("/admin/**").hasRole("ADMIN")

```

```

15     .anyRequest()
16         .authenticated()
17     .and()
18         .cors().disable()
19     .and()
20         .formLogin()
21             .loginProcessingUrl("/do-login")
22             .usernameParameter("email")
23             .passwordParameter("secret")
24             .failureUrl("/login?error=BadCredentials")
25             .defaultSuccessUrl("/home", false)
26     .and()
27         .logout()
28             .logoutUrl("/logout")
29             .clearAuthentication(true)
30             .invalidateHttpSession(true)
31             .deleteCookies("JSESSIONID")
32             .logoutSuccessUrl("/login")
33     .and()
34         .exceptionHandling()
35             .accessDeniedPage("/access-denied");
36     return http.build();
37 }
38 }
```

Изворен код 6.3: Конфигурација на најчесто користените филтри од Spring Security

Вториот бин дефинира синџир со сигурносни филтри, со специфични конфигурации за секој од филтрите. На сликата 6-7 се сумаризирани најважните филтри кои најчесто се користат во рамки на апликациите, заедно со краток опис за секој од нив.

Секој `SecurityFilterChain` бин може да конфигурира различен синџир со сигурносни филтри доколку се постави изразот `.antMatcher(...)`. Со `antMatcher(...)` се конфигурира на кои патеки ќе се активира синџирот со сигурносни филтри кој се конфигурира. Каде ќе се постави оваа конфигурација и во кој редослед се дефинираме со анотацијата `@Order`. Предефинирано, конфигурацијата со редослед `BASIC_AUTH_ORDER` е конфигурирана за патеката `"/**"`. Ова значи дека, ако треба синџирот со филтри да се повика пред предефинираниот, потребно е да се постави помала вредност за редослед, како на пример `BASIC_AUTH_ORDER - 10` (филтер кој што ќе се постави понапред во низата). При заштитата на барањата, се повикува само еден синџир со филтри, а тоа е првиот

кој што ќе има совпаѓање со тековната URL патека на барањето кое се процесира. Ова совпаѓање се прави со `antMatcher` шаблонот, кој исто така се користи при мапирање на сервлетите со URL патеките за кои ќе се повикаат. Во конкретната конфигурација, синцирот со филтри ги опфаќа сите барања кои пристигнуваат до контејнерот.

Аргументот `HttpSecurity` кој се очекува да биде вметнат во `configure` методот се регистрира во контекстот на Spring од страна на `HttpSecurityConfiguration`. Овој објект ја содржи споделената конфигурација која ја прошируваат сите конфигурирани синцири со сигурносни филтри. Со оваа конфигурација се постапуваат предефинирани повици на неговите методи: `csrf()`, `exceptionHandling()`, `headers()`, `sessionManagement()`, `securityContext()`, `requestCache()`, `anonymous()`, `logout()` и `formLogin()`. Секој од овие конфигурациски методи иницијално ги регистрира соодветните филтри во предефинираниот синцир со сигурносни филтри. Доколку има потреба од регистрација на нов, специфично имплементиран филтер, тоа може да се направи во конфигурацијата на синцирите со филтри, преку повик на `http.addFilterBefore(myCustomFilter, SpringSecurityFilter.class)`. На овој начин се регистрира специфичен филтер преку проследување негова инстанца и позиција во синцирот со филтри⁴ релатвна во однос на некој од веќе дефинираните сигурносни филтри. Во конкретниот случај, со проследување на класата на филтерот (`DefaultLoginPageGeneratingFilter.class`), всушност се наоѓа подредувањето на тој проследен филтер и се дефинира дека новиот филтер ќе се постави на позиција пред него.

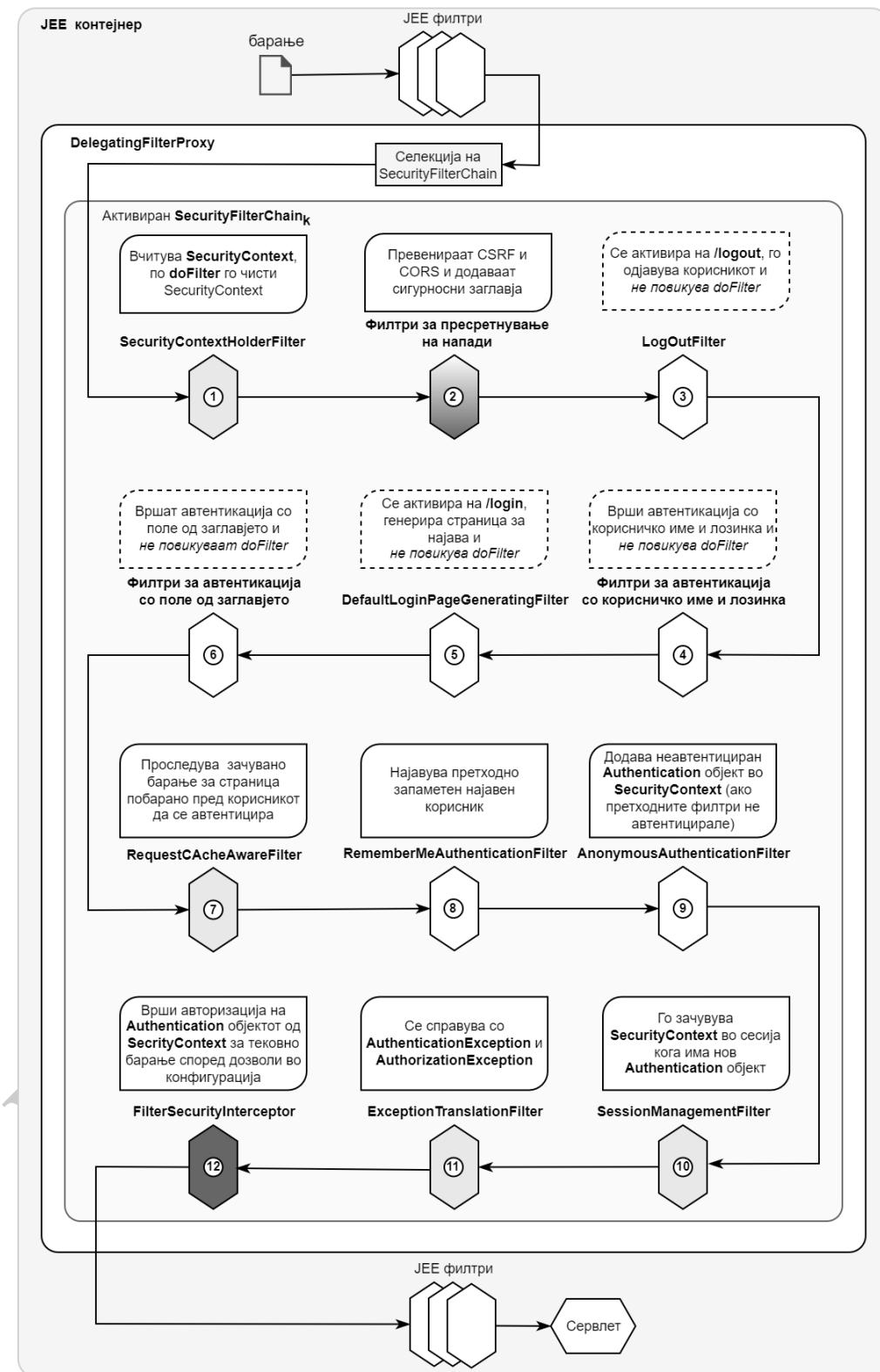
Иницијално барање /admin

За да се објасни процесот на заштита на ресурси, ќе започнеме со барање кон страната `"/admin"` од страна на корисник кој не е најавен. Конфигурацијата од изворниот код 6.3 во линиите 12-16 ја дефинира заштитата на ресурсите (URL патеки во овој случај) и дефинира дека до патеките `"/"`, `"/home"`, `"/register"`, `"/products"` имаат пристап сите корисници, без разлика дали се најавени или не, а додека до патеките со префикс `"/admin"` пристап имаат само најавени корисници со улога `"ADMIN"`. Според оваа конфигурација, барањето до страната `"/admin"` не треба да биде дозволено од страна на Spring Security.

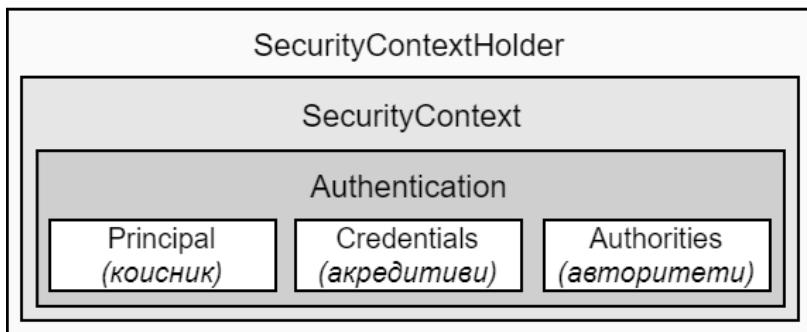
Во овој дел ќе објасниме кои се филтрите кои се активираат⁵ за барањето кон страната `"/admin"` и на кој начин ќе биде забранет пристапот до соодветниот

⁴Може да кажеме и после кој, со користење на `http.addFilterAfter(myCustomFilter, DefaultLoginPageGeneratingFilter.class)`.

⁵Во продолжение ќе дефинираме дека даден сигурносен филтер се активира доколку се изврши логика пред или повик на `chain.doFilter()` методот.



Слика 6-7: Преглед на сигурносните филтри од Spring Security модулот



Слика 6-8: Структура на SecurityContextHolder

ресурс. Иако на сликата 6-7 се дефинирани вкупно 12 филтри, за ова барање ќе бидат активирани само 4 од нив, т.е. филтрите со редни броеви 1, 9, 11 и 12. Останатите филтри нема да се активираат, т.е. ќе се повика само `chain.doFilter()` за нив и нема да се изврши логиката пред и по повикот бидејќи не се исполнети условите за извршување на овој код. Конкретно, сигурносниот филтер 2 е one-возможен во 18-тата линија од конфигурацијата од изворниот код 6.3, а за сите останати филтри, или недостасуваат соодветни параметри (кај филтрите 4, 6, 7, 8 и 10) во барањето, или URL локацијата не одговара на нивните конфигурирани вредности (кај филтрите 3, 4 и 5).

Филтерот со реден број 1, **SecurityContextHolderFilter**, проверува дали постои зачувана верзија од `SecurityContext` и доколку има, истата ја поставува во `SecurityContextHolder`. Потоа, филтерот ја делегира контролата на останатите филтри. Кога тие ќе завршат, тој повторно ја превзема контролата и го чисти контекстот со `SecurityContextHolder.clearContext()` за да се избегне пристап до погрешен контекст при потенцијално реискористување на нишката.

Како што е прикажано на сликата 6-8, интерфејсот `org.springframework.security.core.context.SecurityContext` е местото каде што Spring Security го складира тековниот `Authentication` објект, односно објектот кој го презентира најавениот корисник. За пристап до `SecurityContext` од сите делови на апликацијата се користи класата `org.springframework.security.core.context.SecurityContextHolder`.

`SecurityContextHolderFilter` користи `SecurityContextRepository` за пронаоѓање и зачувување на `SecurityContext`. Предефинираната имплементација го бара `SecurityContext` објектот во сесијата на JEE контejнерот. Може да се промени предефинираното однесување со друга имплементација доколку се дефинира следната конфигурација `http.securityContext().securityContextRepository(mySecCtxRepository);`.

За конкретниот случај, нема да биде пронајден сигурносен контекст и ќе се креира нов, без вредности за `Authentication`, `Credentials` и `Authorities` објектите.

AnonymousAuthenticationFilter (со реден број 9) е наредниот филтер кој се активира. Овој филтер се активира секогаш кога нема автентициран објект во сигурносниот контекст. Ова однесување е потребно за да се додели пристап до ресурси на анонимен корисник, односно корисник кој не е автентициран. Со додавање на анонимен автентициран корисник се дозволув дел од страниците да можат да се пристапат од ненајавени корисници, како што е случајот со страницата за најава.

Со овој филтер се поставува `AnonymousAuthenticationToken` како имплементација на `Authoentication` објектот во `SecurityContext` и со тоа се означува дека моменталното барање е од страна на анонимен корисник. Овој токен може да се користи за да се провери дали еден корисник во рамките на апликацијата е најавен како што е прикажано во изворниот код 6.4. Овој филтер е последен од филтрите за автентикација.

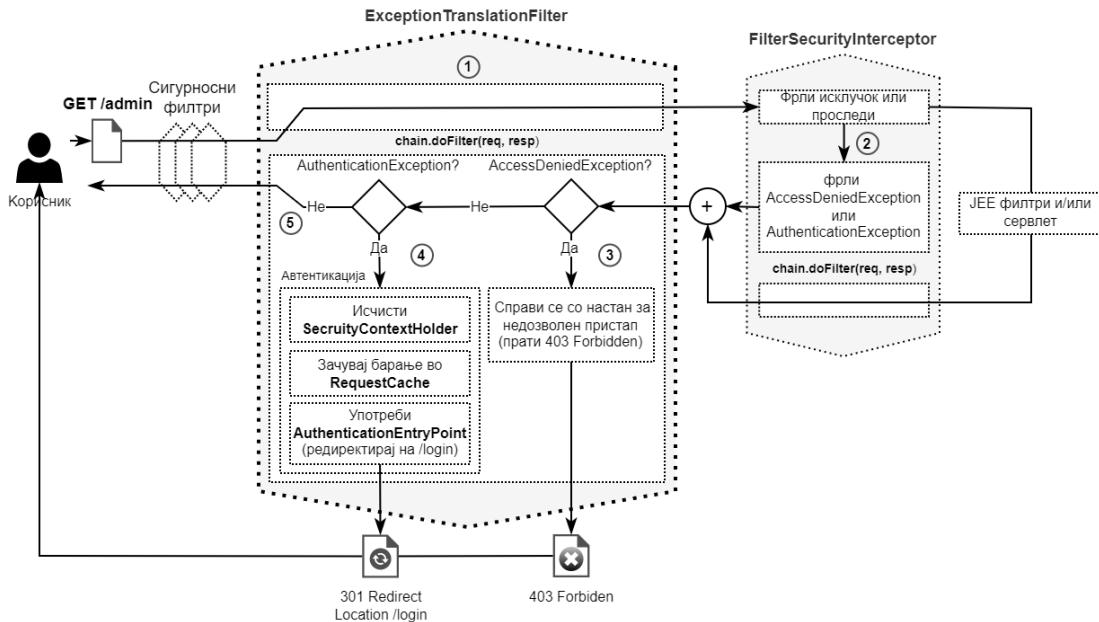
```

1 \begin{listing} [!h]
2 @GetMapping("/")
3 public String method(Authentication authentication) {
4     if (authentication instanceof AnonymousAuthenticationToken) {
5         return "anonymous";
6     } else {
7         return "not anonymous";
8     }
9 }
```

Изворен код 6.4: Проверка дали имаме најавен корисник во рамки на контролер

Нареден се активира филтерот **ExceptionTranslationFilter** (со реден број 11). Овој филтер нема логика пред извршување на методот `chain.doFilter()`, туку целата негова логика се извршува доколку некој од предните филтри генерира исклучок. `ExceptionTranslationFilter` е филтерот кој се справува со исклучоците `AuthenticationException` и `AccessDeniedException` и ги преведува во HTTP одговори. Како што е прикажано на сликата 6-9, `ExceptionTranslationFilter` најпрво повикува `FilterChain.doFilter(request, response)` за да се изврши остатокот од апликацијата. Овој повик е обвиткан во `try-catch` блок. Потоа се справува само со `AuthenticationException` и `AccessDeniedException` исклучоците, а сите останати ги фрла повторно и не се грижи за тие сценарија.

Ова значи дека иницијално, тековното барање се проследува на филтерот **FilterSecurityInterceptor** (со реден број 12). `FilterSecurityInterceptor` - проверува дали тековниот корисник има привилегии да пристапи до ресурси кои ги бара. Конфигурацијата за тоа кои ресурси (во конкретниот случај URL патеки) се достапни за различни корисници се постигнува преку `authorizedRequests()`,



Слика 6-9: Обработка на барања од ExceptionTranslationFilter

како што е прикажано во изворниот код 6.3. Редоследот на специфицирањето на `antMatchers(...)` деловите е важен затоа што за секое барање, совпаѓањата се проверуваат според редоследот на наведување. Ова значи дека првият `antMatchers()` шаблон кој ќе се совпадне со URL патеката на барањето ќе резултира со примена на соодветната дозвола. Така, во примерот од изворниот код 6.3, при повик на патеката `/home` ќе се уважи дозволата `permitAll()`, која означува дека секој корисник може да пристапи до патеката. За конкретното барање кое го разгледуваме, т.е. за патеката `/admin`, ќе се бара автентициран корисник со улога `ADMIN` (дозволата `hasRole("ADMIN")`), а додека пак за патеката `/shopping-cart`, која има совпаѓање со `anyRequest()`, што одговара на сите останати патеки кои не се претходно наведени, доволно ќе биде корисниот да е автентициран (дозволата `authenticated()`). Процесот на авторизација со Spring Security ќе биде описан во повеќе детали во поглавјето 6.4.4.

Во конкретното сценарио, `FilterSecurityInterceptor` ќе увиде дека во сигурносниот контекст `SecurityContext` има анонимен корисник (`AnonymousAuthenticationToken`), а во конфигурацијата за тековното барање `"/admin"` е наведено дека треба да има корисник со улога `"ADMIN"`. Во овој случај ќе биде генериран исклучок од типот `AuthenticationException`.

Во овој случај, `ExceptionTranslationFilter` поставува нов празен `SecurityContext` во `SecurityContextHolder`. Потоа, тековното барање се зачувува во `RequestCache`, со што се овозможува по најавата корисникот да пристапи до ресурсот кој иницијално го побарал. Наредно се повикува кон-

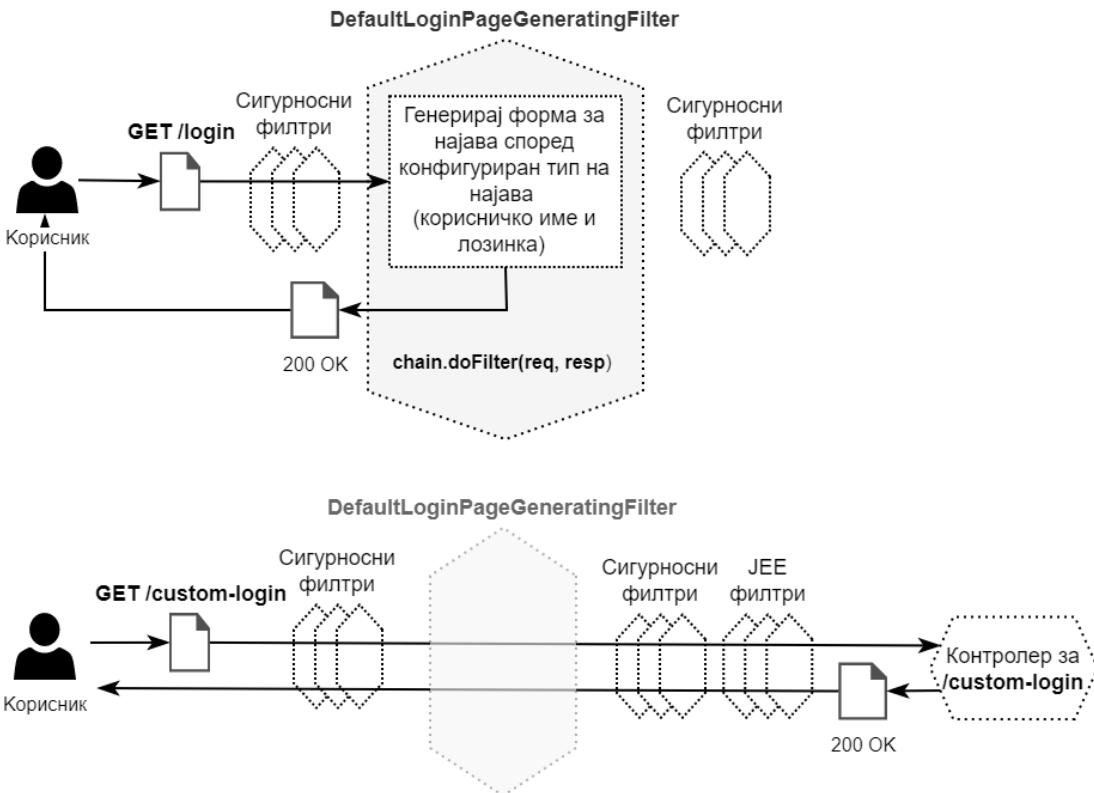
фигурираниот `AuthenticationEntryPoint` за да се побара од корисникот да внесе валидни акредитиви. Имплементациите на `AuthenticationEntryPoint` треба во одговорот да додадат информации, кои во зависност од методот за автентикација, на клиентот ќе му дадат до знаење дека треба да испрати свои акредитиви за најава. Во зависност од конфигурираните методи за автентикација, може корисникот да биде редиректиран за страница за најава преку `LoginUrlAuthenticationEntryPoint` (процес прикажан на слика 6-11), или да се испрати `WWW-Authenticate` заглавјето во одговорот со статусен код 401 `Unauthorized` преку `BasicAuthenticationEntryPoint`.

`AuthenticationEntryPoint` имплементациите се регистрираат за `ExceptionTranslationFilter` во процесот на конфигурација, при што за секоја од овие имплементации треба да се проследи и `RequestMatcher` имплементација, која ќе каже за кои параметри од барањето ќе се активира соодветниот `AuthenticationEntryPoint`. Така, со вклучување на конфигурацијата `formLogin()`, автоматски се вклучува `LoginUrlAuthenticationEntryPoint` за барањата кои во `Accept` заглавјето содржат некои од типовите `application/xhtml+xml`, `text/html`, `text/plain` или `image/*`. Слично, вклучувањето на основната автентикација преку `httpBasic()` автоматски го регистрира `BasicAuthenticationEntryPoint` за барањата кои прифаќаат некои од типовите: `MediaType.APPLICATION_ATOM_XML`, `MediaType.APPLICATION_FORM_URL_ENCODED`, `MediaType.APPLICATION_JSON`, `MediaType.APPLICATION_OCTET_STREAM`, `MediaType.APPLICATION_XML`, `MediaType.MULTIPART_FORM_DATA`, `MediaType.TEXT_XML`, `MediaType.APPLICATION_ATOM_XML`, `MediaType.APPLICATION_FORM_URL_ENCODED`, `MediaType.APPLICATION_JSON`, `MediaType.APPLICATION_OCTET_STREAM`, `MediaType.APPLICATION_XML`, `MediaType.MULTIPART_FORM_DATA`, `MediaType.TEXT_XML`.

При справувањето со `AccessDeniedException`, се повикува регистрираниот `AccessDeniedHandler` за да се спрavi со забраната за пристап. Предефинираната имплементација на `AccessDeniedHandler` единствено постапува на одговорот статусен кодо 403 `Forbidden`. Предефинираното однесување може да се конфигурира со повик на `http.exceptionHandling().accessDeniedPage("/access-denied")` кој ќе овозможи да се повика контролерот кој се справува со патеката `/access-denied` и таму да дефинира специфичен приказ на грешката која настанала.

Редирекција кон `/login`

Бидејќи барањето кон страницата `"/admin"` беше од не-автентициран корисник, `ExceptionTranslationFilter` извршува редирекција кон конфигурираната патека за автентикација. Бидејќи ова не е специфично конфигурирано во изворниот



Слика 6-10: Обработка на барања од DefalutLoginPageGeneratingFilter за сценарија без и со конфигурирана предефинирана патека за логирање

код 6.3, се извршува редирекција кон патеката "[/login](#)".

По добивањето на редиректованото барање со патека "[/login](#)", повторно се извршува синцирот со филтри конфигуриран во изворниот код 6.3, но во овој случај се активираат само филтрите 1 и 5.

`SecurityContextHolderFilter` (со реден број 1) филтерот ќе иницијализира празен сигурносен контекст како и во претходното барање, бидејќи претходно не се случила успешна автентикација и не е зачувано ништо.

`DefaultLoginPageGeneratingFilter` (со реден број 5) е филтер кој креира предефинирана страницата за најава ако корисникот не е најаваен. Овој филтер предефинирано ги пресретнува HTTP барањата од типот GET на патеката `/login` и ќе се активира за тековното барање. Филтерот знае како да генерира интерфејс за најава за сите овозможени филтри за автентикација. Доколку се активира, тој го гради телото на одговорот и не ги повикува наредните филтри.

Доколку е потребно да се постави различен од предефиниријаниот изглед на страницата, тоа може да се постигне преку `http.formLogin().loginPage("/custom-login")`. На овој начин, секогаш кога ќе биде потребна автентикација, `ExceptionTranslationFilter` ќе редиректира на

патеката `/custom-login`, при што ќе се повика методот од контролерот задолжен со спроведување со барањата на оваа патеката, кој ќе треба да го генерира изгледот на формата за автентификација.

За тековната конфигурација, ќе се креира форма за автентификација со корисничко име и лозинка. Оваа форма ќе генерира POST барање кон локацијата конфигурирана со `.loginProcessingUrl("/do-login")`, или кон `"/login"` доколку немаме експлицитна конфигурација. Дополнително, имињата на влезните полиња ќе бидат поставени според конфигурациите `.usernameParameter("email")` (предефинирано `"username"`) за корисничкото име и `.passwordParameter("secret")` (пре-дефинирано `"password"`) за лозинката.

Автентификација со корисничко име и лозинка (POST `/login`)

По приказот на формата за автентификација, корисникот може да ги внесе своите акредитиви (корисничко име и лозинка). Доколку внесените податоци се погрешни, при испраќање на формата (при генерирање на барањето POST `/login`) ќе се добие одговор кој ќе не пренасочи на страницата конфигурирана со `.failureUrl("/login?error=BadCredentials")`, која во овој случај е истата страница, но со дополнителен параметар кој означува дека акредитивите не се валидни.

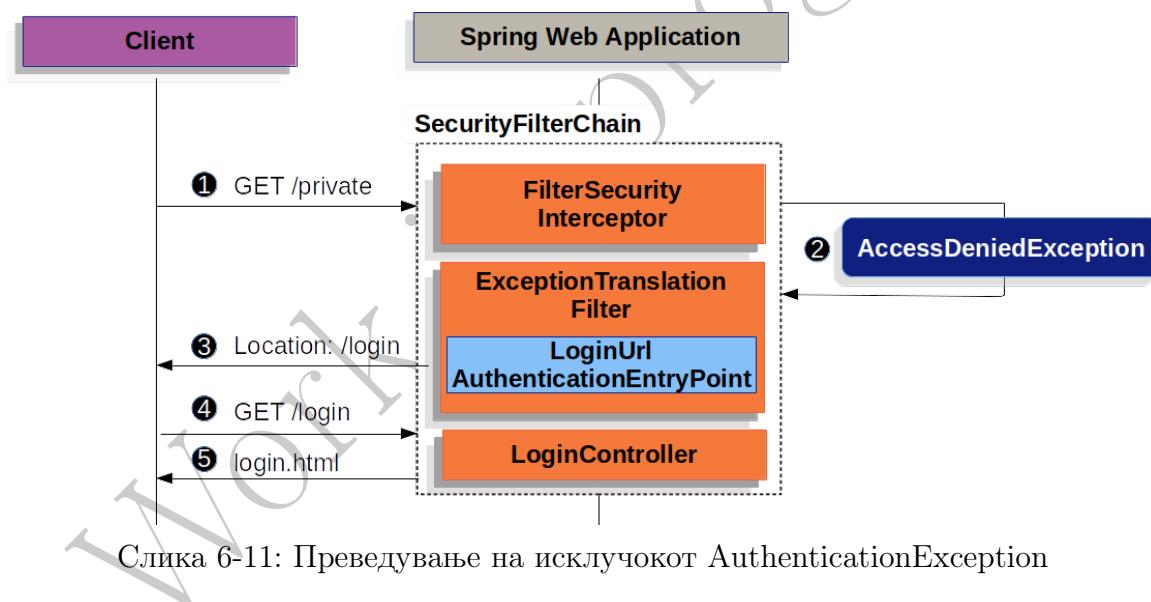
Во случај на внес на валидни акредитиви, се активираат филтрите 1 и 4.

`SecurityContextHolderFilter` повторно ќе иницијализира празен сигурносен контекст, бидејќи сеуште нема успешно автентициран корисник.

Филтерот `UsernamePasswordAuthenticationFilter` (со реден број 4) се активира, бидејќи тековното HTTP барање е од типот POST на патеката `"/login"`, по што ги бара параметрите со имиња `username` и `password` во телот на барањето и со нивна помош се обидува да го автентицира корисникот кој го испратил барањето. Предефинираните имиња на својства може да се променат како што е прикажано во изворниот код [6.3](#). Во случај на успешна најава, `UsernamePasswordAuthenticationFilter` добива валиден `Authentication` објект и ги прави следните акции:

- `Authentication` објектот го поставува во тековниот `SecurityContext` (добиен од `SecurityContextHolder`)
- Го зачувува тековниот сигурносен контекст преку `SecurityContextRepository` (предефинирано во сесијата на JEE контек-јнерот)
- Доколку е конфигурирано да се запомнат автентицираните корисници, се зачува тековниот `Authentication` објект со помош на `RememberMeService`. Во тековната конфигурација, нема да биде запаметен корисникот, бидејќи не е конфигуриран овој сервис.

- Се врши редирекција на страната за успешна најава, која може да се конфигурира со `defaultSuccessUrl("...", true/false)`. Овде се проследува и втор аргумент кој означува дали корисникот секогаш да биде пренасочен на конфигурираната патека со првиот аргумент. Во изворниот код 6.3 е конфигурирано дека при успешна најава ќе се изврши редирекција на патеката за која била потребна автентикација, пред редирекцијата на страната за најава. Доколку директно е пристапено на страната за најава и нема запаметено (кеширано) претходно барање, ќе се изврши редирекција на конфигурираната патека. Во овој случај, по успешна најава, ако вториот аргумент е поставен на `false` (предефинирана вредност), `UsernamePasswordAuthenticationFilter` ќе го замени тековното барање со претходно зачуваното барање до првично посакуваната страна `"/admin"`. Ако вредноста на вториот аргумент е `true`, по успешната најава, корисникот секогаш ќе биде пренасочен на конфигурираната патека (во примерот, `"/home"`). Дополнително објаснување за начинот на кој функционира процесот на автентикација е дадено во поглавјето 6.4.3.

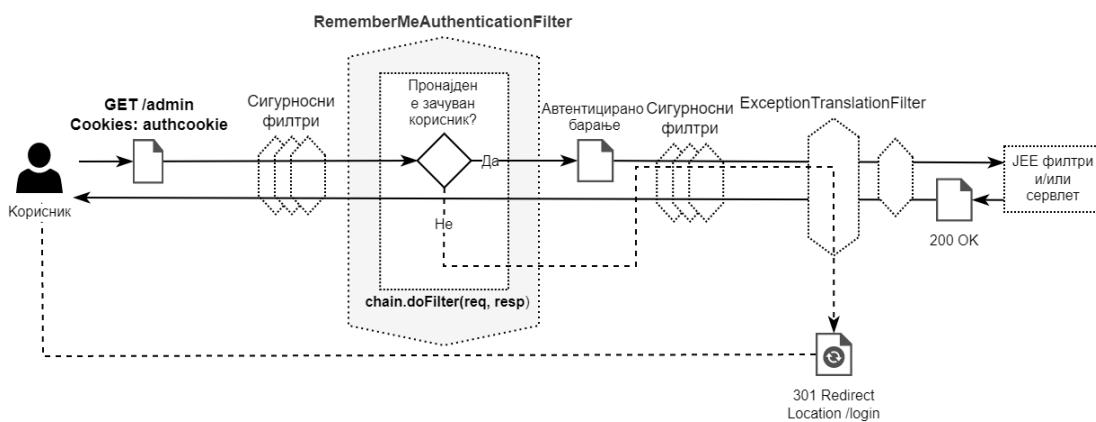
Слика 6-11: Преведување на исклучокот `AuthenticationException`

Редирекција кон `/admin/products`

`RequestCacheAwareFilter` е филтер кој го памети барањето за страницата со ограничен пристап до која се обидува да пристапи неавтентициран корисник. Барањето го сместува во `RequestCache` и, откако корисникот успешно ќе се најави, го проследува за понатамошна обработка. Се оневозможува со поставување на `true` за втор аргумент на методот `defaultSuccessUrl`

Барање кон /admin/products за запаметен корисник

RememberMeAuthenticationFilter е дел од филтрите за автентикација, кој се обидува да пронајде дали на некој начин е зачувана најавата, за автоматски да го најави корисникот. Филтерот користи имплементација на **RememberMeServices** интерфејсот за да го пронајде автентицираниот корисник. Истиот овој интерфејс се користи и од **AbstractAuthenticationProcessingFilter** (филтерот од кој наследува **UsernamePasswordAuthenticationFilter**) за да го зачува најавениот корисник.⁶



Слика 6-12: Обработка на барања од RememberMeAutehenticationFilter

Одјава на корисник

LogOutFilter се справува со одјава на корисниците. Предефинирано, овој филтер се повикува на патеката `/logout`. Доколку се повика URL патеката со која се активира овој филтер, ќе се повикаат конфигурираните опции со кои се одјавува корисникот, по што се генерира одговор кој ќе го пренасочи корисникот на страница која е конфигурирана да се прикаже при успешна одјава.

Во изворниот код [6.3](#) е прикажан начинот на кој може да се промени URL патеката на која се активира овој филтер преку методот `logoutUrl`. Дополнително, во примерот се прикажани и други функционалности кои може да се извршат во рамките на овој филтер како што се отстранување на најавениот корисник од `ServletContext` преку `clearAuthentication(true)` методот, инвалидирање на сесијата преку `invalidateHttpSession(true)`, отстранување на специфични колачиња преку `deleteCookies(...)` и поставување на URL патека на која ќе се пренасочи одјавениот корисник преку `(logoutSuccessUrl("..."))`.

⁶Повеќе детали за конфигурацијата на овој филтер може да добиете на следниот линк: <https://docs.spring.io/spring-security/reference/servlet/authentication/rememberme.html>

Наредниот филтер кој се активира е **RequestCacheAwareFilter** (со реден број 7 да слика 6-7). Овој филтер го памети барањето за страницата со ограничен пристап до која се обидува да пристапи неавтентициран корисник. Барањето го сместува во **RequestCache**, па откако корисникот успешно ќе се најави, го проследува за понатамошна обработка. Се оневозможува со поставување на **true** за втор аргумент на методот **defaultSuccessUrl**.

6.4.2 Синцир со сигурносни филтри

При стартирање на апликација, најпрво се вчитуваат безбедносните конфигурации на Spring Security . За да се читаат, конфигурациите треба да се регистрирани како бинови во рамките на Spring IoC контекстот. Подоцна, при HTTP барање, прво се избира синцирот со филтри кој е применлив за тоа барање, по што се процесираат сигурносните филтри од синцирот според нивното подредување дефинирано во Spring Security модулот. Во Spring Boot пости голем број на предефинирани филтри кои се извршуваат при обработка на секое барање. Дел од тие филтри ⁷ кои се особено значење за имплементација на безбедноста се прикажани според редоследот на нивното повикување се:

1. **Филтри за пресретнување на напади.** Во оваа група на филтри припаѓаат **CorsFilter**, **CsrfFilter** и **HeaderWriterFilter** филтрите. **CorsFilter** бара да биде присутен токен во заглавјето или параметар во барањето кој се валидира во однос на зачуваниот токен. Зачуваниот токен се добива со помош на имплементација на **CsrfTokenRepository**. Доколку токенот не е валиден, барањето се третира како да нема привилегии, односно резултира со одговор со статус код 403 (Forbidden). **CorsFilter** проверува дали е дозволено да се примаат барања од локацијата проследена во *Origin* заглавјето. И двата филтри се дизајнирани да спречат извршување на остатокот од филтрите и на самата апликација доколку не се исполнети условите дефинирани во нив. Последниот филтер, **HeaderWriterFilter**, додава заглавја во одговорот со кои се зголемува безбедноста ⁸.
2. **Филтри за автентикација** - филтри кои може да извршат автентикација на корисникот со различни механизми. Дел од регистрираните филтри за автентикација, според редоследот на извршување се: **OAuth2AuthorizationRequestRedirectFilter**, **Saml2WebSSoAuthenticationRequestFilter**, **X509AuthenticationFilter**, **AbstractPreAuthenticatedProcessingFilter**, **CasAuthenticationFilter** и

⁷Целосната листа на филтри според редоследот може да се види на: <https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-security-filters>

⁸<https://docs.spring.io/spring-security/reference/features/exploits/headers.html>

`UsernamePasswordAuthenticationFilter`. Од овие филтри, подетално ќе го разгледаме само `UsernamePasswordAuthenticationFilter` кој овозможува автентикација со корисничко име (username) и лозинка (password).

3. **Филтрите за автентикација со податок од заглавје** се справуваат со најава за која `Authentication` објектот се гради со податок од заглавјето. Такви филтри се `DigestAuthenticationFilter`, `BearerTokenAuthenticationFilter` и `BasicAuthenticationFilter`. Подетално ќе го објасниме само `BasicAuthenticationFilter`, кој го следи протоколот за основна автентикација (Basic Authentication). Кога се користи основна автентикација, корисниците вклучуваат енкодиран текст во `Authorization` заглавјето на секое од нивните барања. `Authorization` заглавјето треба да е во форматот `Authorization: Basic <credentials>`. Филтерот `BasicAuthenticationFilter` се активира доколку заглавјето е присутно и доколку е во зададениот формат. Кога ова е случај, филтерот ги декриптира податоците означени со `<credentials>` и на тој начин ги добива корисничко име и лозина. Потоа, најавата продолжува на идентичен начин како кај `UsernamePasswordAuthenticationFilter`.
4. **SessionManagementFilter** е филтер кој проверува дали постои зачуван `ServletContext` преку `SecurityContextRepository`. Во случај кога не постои, а `SecurityContextHolder.getContext().getAuthention()` враќа автентициран објект, овој филтер го зачува контекстот со користење на `SecurityContextRepository.saveContext` методот. Потоа ја делегира контролата на наредните филтри.

6.4.3 Автентикација со Spring Security

Во овој дел најпрво ќе ги разгледаме некои од најважните компоненти кои се користат во процесот на автентикација со Spring Security, по што ќе погледнеме како тие се користат за да се автентицираат корисниците.

Authentication, SecurityContext и SecurityContextHolder

Објектот `Authentication` е апстракција што го претставува ентитетот што се најавува во системот. Најчесто, тоа е корисник. Објектот `Authentication` се користи и кога се креира барање за автентикација (кога корисникот се најавува), за да ги пренесе податоците што се потребни низ различните слоеви и класи од апликацијата и Spring Security модулот. Ако автентикацијата е успешна, овој објект го содржи автентицираниот корисник и го складира во `SecurityContext`.

Во типично сценарио на безбедна апликација, кога корисникот ќе се најави, се креира посебен објект за автентикација што ги складира кориснич-

кото име (*Principal*), акредитациите (*credentials*) и авторитетите (*Authorities*). Интерфејсот `Authentication` е прилично едноставен и е прикажан во изворниот код [6.5](#). Преку методот `isAuthenticated()` може да се направи разлика дали `Authentication` објектот е изграден од некој од филтрите со цел да ги пренесе акредитивите за најава или е резултат од успешна автентикација вратена од `AuthenticationManager`. Методот `setAuthenticated` се користи од страна на `AuthenticationManager` за означување дека објектот е валидно автентициран. Својството `principal` означува кој е најавениот корисник. При најава со корисничко име и лозинка, ова својство е инстанца од `UserDetails`. Својството `credentials` се однесува на акредитивите, но најчесто по успешна автентикација, ова својство се брише, за да не се чуваат сензитивни податоци подолго од потребното. Доделените авторитети (`GrantedAuthority`) се дозволи кои ги добива корисникот на високо ниво. Најдобар пример за доделени авторитети се улогите. Кај автентикација со корисничко име и лозинка, `GrantedAuthority` обично се читуваат од `UserDetailsService`. Овие доделени авторитети се користат во конфигурациите за авторизацијата, каде може да се наведе за кои улоги се дозволени или забранети одредени ресурси.

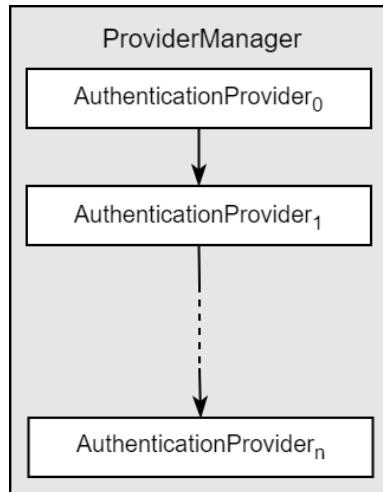
```

1 public interface Authentication extends Principal, Serializable {
2     Collection<? extends GrantedAuthority> getAuthorities();
3     Object getCredentials();
4     Object getDetails();
5     Object getPrincipal();
6     boolean isAuthenticated();
7     void setAuthenticated(boolean isAuthenticated) throws
        IllegalArgumentException;
8 }
```

Изворен код [6.5: Authentication](#)

Интерфејсот `org.springframework.security.core.context.SecurityContext` е местото каде што Spring Security го складира тековниот `Authentication` објект, поврзувајќи го со тековната нишка која го процесира барањето. Класата што се користи за пристап до `SecurityContext` од сите делови на апликацијата е `org.springframework.security.core.context.SecurityContextHolder`. Оваа класа е изградена главно од статични методи за складирање и пристап до `SecurityContext`. Структурата на `SecurityContextHolder` е прикажана на слика [6-8](#).

AuthenticationManager и AuthenticationProvider



Слика 6-13: Структура на ProviderManager

```

1 public interface AuthenticationManager {
2     Authentication authenticate(Authentication authentication) throws
3         AuthenticationException;
4 }
  
```

Изворен код 6.6: AuthenticationManager

`AuthenticationManager` е интерфејс кој дефинира како се врши автентикацијата од страна на сигурносните филтри. Неговата структура е прикажана во изворниот код 6.6. Се состои од методот за автентикација `authenticate` кој враќа објект `Authentication`, кој подоцна филтрите за автентикација го додаваат во `SecurityContextHolder`. Доколку автентикацијата не е успешна, методот фрла исклучок од типот `AuthenticationException`. Предефинираната имплементација на интерфејсот `AuthenticationManager` е `ProviderManager`. Како што е прикажано на слика 6-13, `ProviderManager` содржи листа од `AuthenticationProvider` имплементации. Почнувајќи од првата `AuthenticationProvider` имплементација, се изминува секоја наредна имплементација од листата сè додека некоја од нив не изврши успешна автентикација.

`AuthenticationProvider` е главната влезна точка за автентикација на објектот `Authentication`. Овој интерфејс има два методи, прикажани во изворниот код 6.7.

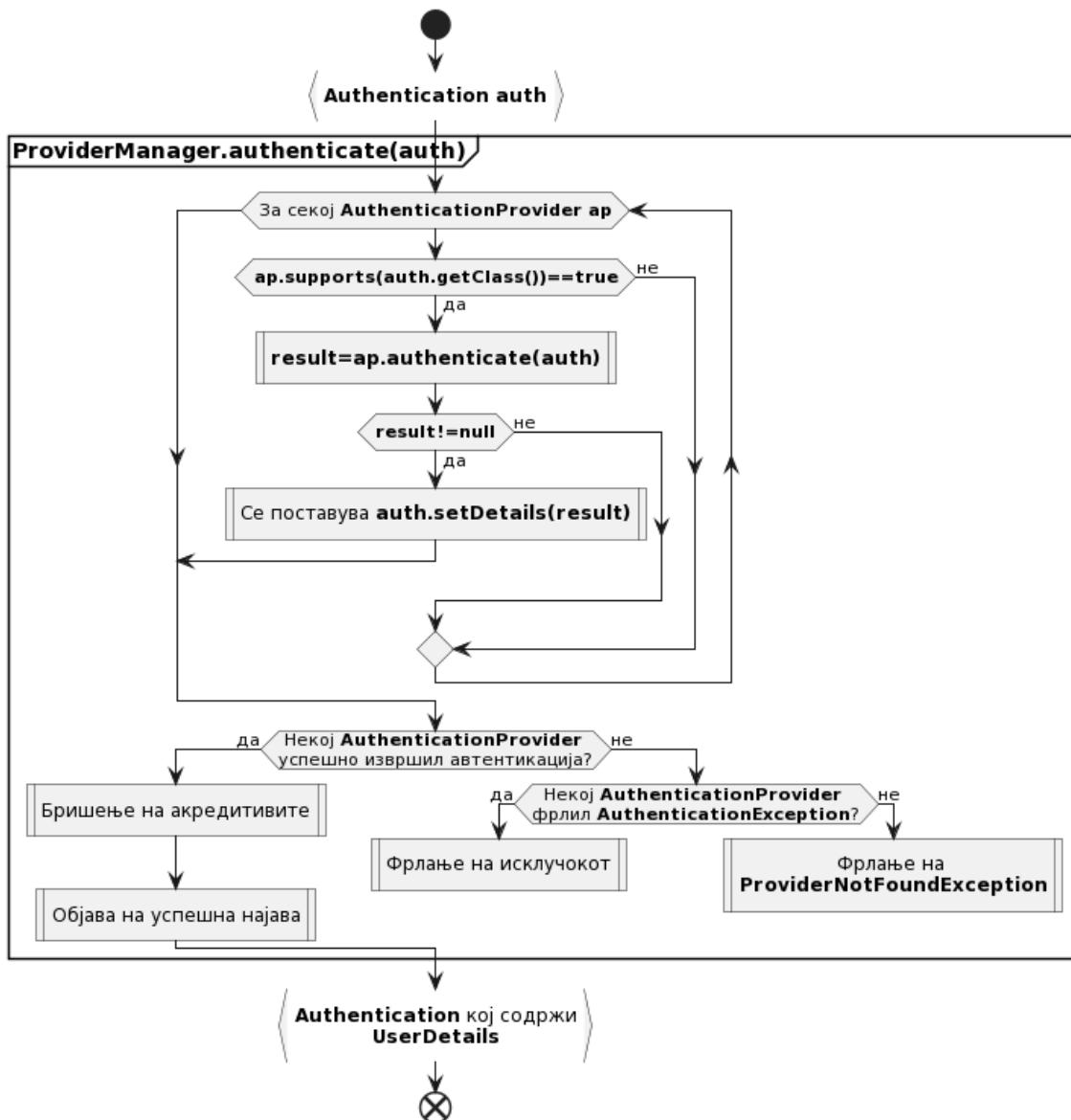
```

1 public interface AuthenticationProvider {
2     Authentication authenticate(Authentication authentication) throws
3         AuthenticationException;
4     boolean supports(Class<?> authentication);
5 }
  
```

Изворен код 6.7: AuthenticationProvider

На сликата [6-14](#) е прикажан целокупниот процес на автентикација преку `ProviderManager`. Откако ќе пристигне автентикациски објект кој содржи акредитиви, `ProviderManager` ги изминува имплементациите `AuthenticationProvider` така што за секој од нив најпрво го повикува методот `supports(...)` за да порвери дали имплементацијата на тековниот `AuthenticationProvider` знае да изврши автентикација со податоците проследени во `Authentication` објектот. Доколку е соодветен `AuthenticationProvider`-от, се повикува неговиот `authenticate` метод, кој треба да ги провери акредитивите на проследениот автентикациски објект. Ако автентикацијата е успешна, се враќа `Authentication` објект на кој му се поставени деталите за корисникот. Ако акредитивите не се валидни, се фрла `AuthenticationException`. Доколку `AuthenticationProvider`-от не може да одлучи дали акредитивите се валидни или не, тој може да врати `null`, со што `ProviderManager` ќе продолжи со процесирање на следниот регистриран `AuthenticationProvider`. Ако никој не еден од конфигурираните `AuthenticationProvider`-и не може да изврши автентикација, `ProviderManager` го фрли исклучокот `ProviderNotFoundException`, кој наследува од `AuthenticationException` со кој се дава до знаење дека нема `AuthenticationProvider` со кој може да се автентицира тековниот `Authentication` објект. При успешна автентикација, `ProviderManager` ќе се обиде да ги избрише сензитивните акредитиви кои се чуваат во `Authentication` објектот добиен по успешната автентикација. Ова спречува сензитивните информации, како што е лозинката, да се задржат подолго отколку што е потребно во сесијата или на други места кај што се зачувува овој објект. За да се оневозможи бришење на акредитивите во овој процес, се поставува `auth.eraseCredentials(false)` во методот `configure(AuthenticationManagerBuilder auth)` на конфигурацијата која наследува од `WebSecurityConfigurerAdapter`. Иако `WebSecurityConfigurerAdapter` е маркирана како `@Deprecated`, во одредени случаи ќе го користиме овој начин на конфигурација, затоа што сеуште новиот начин не е целосно во употреба, а и сеуште има голем број на апликации кои се конфигурирани со `WebSecurityConfigurerAdapter`. Класите кои наследуваат од `WebSecurityConfigurerAdapter` треба да бидат вметнати во контекстот за инверзија на контролата на Spring за да се земат предвид. Методот `configure(HttpSecurity http)` е стариот начин за конфигурација на `SecurityFilterChain`.

Spring Security модулот поддржува различни имплементации на `AuthenticationProvider` за различни постоечки протоколи за автентикација. Меѓу поважните се вбројуваат:



Слика 6-14: Автентикација со ProviderManager

- CasAuthenticationProvider - провајдер кој користи UsernamePasswordAuthenticationToken за најава на централен сервер за автентикација (Central Authentication Server - CAS). ⁹
- DaoAuthenticationProvider овозможува автентикација на UsernamePasswordAuthenticationToken со помош на UserDetailsService и PasswordEncoder. PasswordEncoder имплементацијата овозможува чување на енкриптирана верзија од лозинката и проверка дали проследената лозинка одговара на зачуваната енкриптирана верзија.

⁹<https://docs.spring.io/spring-security/reference/servlet/authentication/cas.html>

- `OpenIDAuthenticationProvider` овозможува најава со користење на протоколот OpenID.¹⁰
- `RememberMeAuthenticationProvider` користи `RememberMeAuthenticationToken` кој е генериран од страна на `TokenBasedRememberMeServices` во рамките на `RememberMeFilter` за да го автентицира корисникот на тековното барање.¹¹
- `LdapAuthenticationProvider` овозможува најава преку LDAP протоколот со користење на `UsernamePasswordAuthenticationToken`.

Ако наидеме на одреден протокол за автентикација на корисници кој не е поддржан од модулот SpringSecurity, веројатно ќе треба да го имплементираме овој интерфејс со потребната функционалност. Затоа, `AuthenticationProvider` е една од главните точки за проширување на модулот Spring Security. Конфигурација на нов `AuthenticationProvider` се постигнува преку `http.authenticationProvider(myAuthProvider)` во конфигурацијата за синцирот со сигурносни филтри.

UserDetailsService и UserDetails

Имплементациите на интерфејсот `UserDetailsService` се задолжени за вчитување на корисничките информации од складиштето на корисници (зачувани во меморија, во база на податоци или на друго место) при процесот на автентикација на барањето кое пристигнува во апликацијата. `UserDetailsService` го користи добиеното корисничко име за пребарување на остатокот од потребните кориснички податоци од складиштето на корисници. Овој интерфејс дефинира само еден метод и неговата дефиниција е дадена во изворниот код 6.8. Може да се конфигурира и кориснички дефинирана имплементација на `UserDetailsService` со едноставно нејзино додавање во апликацискиот контекстот за инверзија на контролата.

```

1 public interface UserDetailsService {
2     UserDetails loadUserByUsername(String username) throws
3         UsernameNotFoundException;

```

Изворен код 6.8: `UserDetailsService`

Интерфејсот `org.springframework.security.core.userdetails.UserDetails` е апстракција која се користи за претставување на целосен корисник во контекст на Spring Security. До деталите за корисникот може да се пристапи по неговата најава преку

¹⁰<https://docs.spring.io/spring-security/reference/servlet/authentication/openid.html>

¹¹<https://docs.spring.io/spring-security/reference/servlet/authentication/rememberme.html>

`SecurityContextHolder.getContext().getAuthentication().getPrincipal()`. Вообично, се дефинира сопствена имплементација на овој интерфејс за да складираат дополнителни кориснички детали како што се е-пошта, телефон, адреса и слично. Изворниот код 6.9 го прикажува интерфејсот `UserDetails`. Покрај корисничкото име, лозинката и доделените авторитети, овој интерфејс дефинира и методи за проверка на сметката на корисникот.

```

1 public interface UserDetails extends Serializable {
2     Collection<? extends GrantedAuthority> getAuthorities();
3     String getPassword();
4     String getUsername();
5     boolean isAccountNonExpired();
6     boolean isAccountNonLocked();
7     boolean isCredentialsNonExpired();
8     boolean isEnabled();
9 }
```

Изворен код 6.9: `UserDetails`

Некои од имплементациите на `UserDetailsService` (на пример, `InMemoryUserDetailsManager`) ја користат класата `org.springframework.security.core.userdetails.User` како имплементацијата на `UserDetails` за резултат од методот `loadUserByUsername`. Сепак, ова е уште една од оние точки на Spring Security што може да се конфигурираат и лесно може да креираме своја сопствена имплементација на `UserDetails` и да ја користиме како резултат на `UserDetailsService` во нашата апликација.

Покрај `UserDetailsService`, постои и интерфејс `AuthenticationUserDetailsService` кој е погенерички и овозможува да се добие `UserDetails` користејќи `Authentication` објект, наместо корисничко име од тип `String`. Оваа функционалност го прави пофлексибilen за имплементација на автентификацијата. Неговата дефиниција е прикажана во изворниот код 6.10.

```

1 public interface AuthenticationUserDetailsService<T extends Authentication>
2 {
3     UserDetails loadUserDetails(T token) throws UsernameNotFoundException;
}
```

Изворен код 6.10: `AuthenticationUserDetailsService`

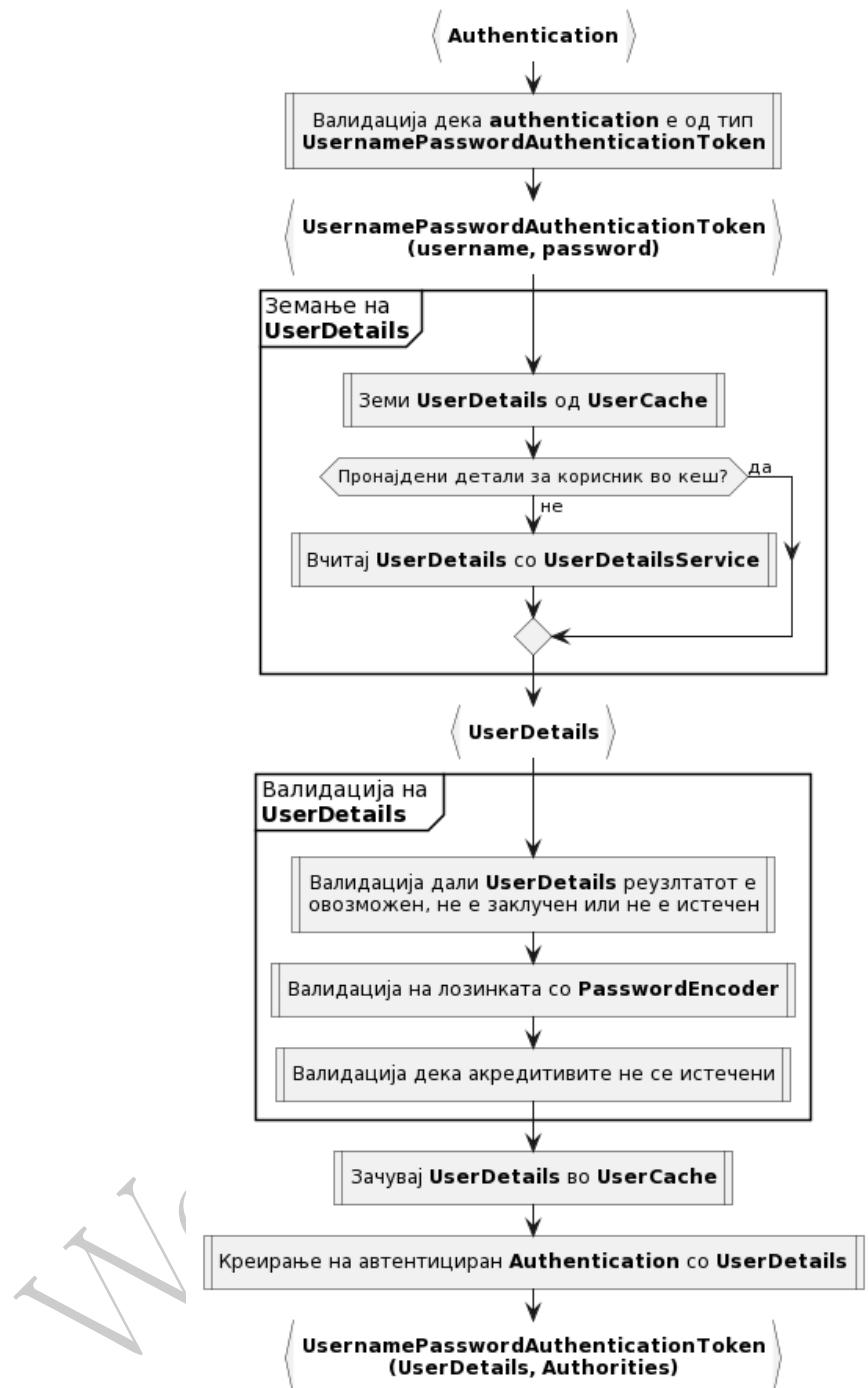
Интерфејсите `AuthenticationUserDetailsService` и `UserDetailsService` се двете главни стратегии кои се користат за враќање на корисничките информации при обид за автентификација. Најчесто тие се повикуваат од имплемента-

цијата на `AuthenticationProvider` што се користи во апликацијата. На пример, `OpenIDAuthenticationProvider` и `CasAuthenticationProvider` ја делегираат задачата да ги добијат деталите за корисникот на `AuthenticationUserDetailsService`, додека `DaoAuthenticationProvider` ја делегира оваа задача на `UserDetailsService`. Некои други провајдери не користат никаков вид услуга за детали за корисникот (на пример, `JaasAuthenticationProvider` користи свој механизам за да го врати `Principal` од `javax.security.auth.login.LoginContext`), а некои други користат целосно приспособена имплементација (на пример, `LdapAuthenticationProvider` користи `UserDetailsContextMapper`).

DaoAuthenticationProvider

`DaoAuthenticationProvider` е најчесто користениот `AuthenticationProvider` кој врши најава со користење на крисничко име и лозинка. На сликата 6-15 е прикажано како функционира методот `DaoAuthenticationProvider.authenticate()`. Во првиот чекор се валидира дали проследениот `Authentication` објект е од типот `UsernamePasswordAuthenticationToken`. Ако не е, се фрла исклучок од типт `IllegalArgumentException`. Потоа, се проверува дали има претходно зачувана објект `UserDetails` за тековното корисничко име, добиено со методот `UsernamePasswordAuthenticationToken.getPrincipal()`. За да воопшто се зачувуваат овие инстанци во кеш, потребно е експлицитно да се конфигурира имплементација на `UserCache`. Предефинирано се користи `NullUserCache` имплементацијата која не зачува детали за корисници и секогаш враќа `null` како резултат. Во случај кога не се пронајдени детали за корисникот во кешот, се повикува `loadUserByUsername` од конфигурираниот `UserDetailsService`. Ако ресултатот на овие операции е фрлен исклучок или резултат од типот `null`, автентификацијата прекинува и исклучокот се фрла и од `DaoAuthenticationProvider`-от.

Откако ќе се добијат деталите за корисникот, истите треба да се валидираат. `DaoAuthenticationProvider` најпрво ја валидира состојбата на корисничката сметка, односно дали корисникот е овозможен (`UserDetails.isEnabled()`), дали не е заклучен (`UserDetails.isAccountNonLocked()`) и дали сметката не е истечена (`UserDetails.isAccountNonExpired()`). Доколку кој било од овие услови не е исполнет, се фрла соодветна верзија на `AccountStatusException`, која пак е изведена од `AuthenticationException`. Наредно се проверува дали лозинката која е испратена во барањето (достапна преку `UsernamePasswordAuthenticationToken.getCredentials()`) е иста со лозинката од добиените детали за корисникот. Притоа, во овој процес се бара лозинките кои ќе се добијат во детлите за корисникот да бидат енкриптирани, со што се наложува поголема заштита на овие приватни податоци. Затоа, за проверката се



Слика 6-15: DaoAuthenticationProvider

користи PasswordEncoder со кој лозинката добиена од барањето прво се енкриптира, па се споредуваат енкриптирани верзии. Во DaoAuthenticationProvider веќе има предефиниран PasswordEncoder, но истиот може да се замени со регистрирање на бин во контекстот кој го имплементира PasswordEncoder

интерфејсот. Доколку лозинката не е валидна, се фрла `BadCredentialsException`. Доколку лозинката е валидна, се проверува уште и дали таа не е истечена преку `UserDetails.isCredentialsNonExpired()`. Доколку е истечена се фрла `CredentialsExpiredException`.

Ако сите валидации се успешни, се зачувуваат корисничките детали во `UserCache` и се гради нов `UsernamePasswordAuthenticationToken` кој ги содржи `UserDetails` и доделените авторитети кои тој ги содржи. Притоа, авторитетите од `UserDetails` се конвертираат со `GrantedAuthoritiesMapper` пред поставувањето во `UsernamePasswordAuthenticationToken`. Предефинираната имплементација на `GrantedAuthoritiesMapper` е `NullAuthoritiesMapper`, која не врши никаква трансформација и ги враќа оригиналните доделени авторитети.

AbstractAuthenticationProcessingFilter

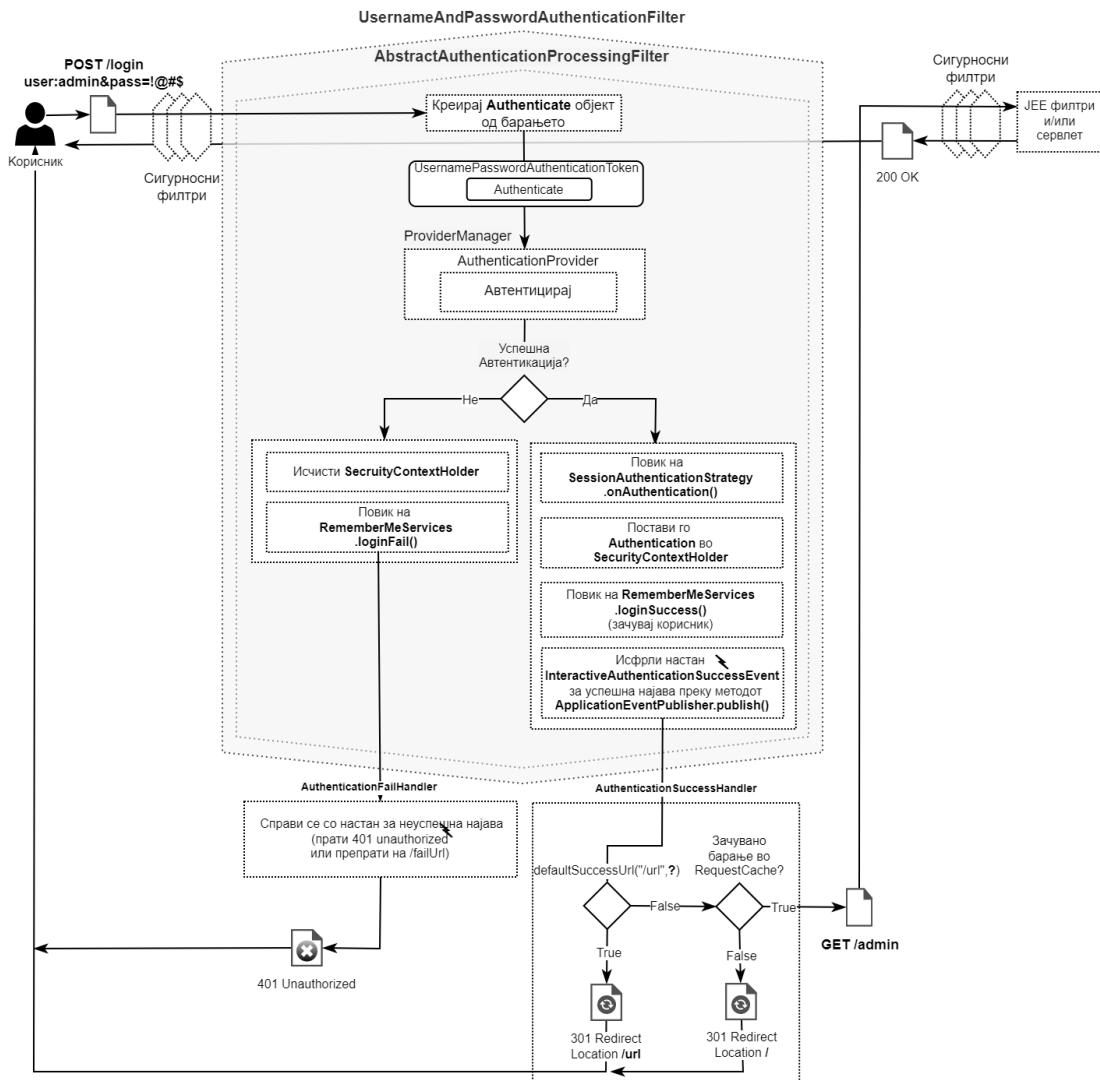
`AbstractAuthenticationProcessingFilter` се користи како основен филтер во процесот на автентикација на корисниците. Овој филтер се справува со генералниот процес на автентикација, без разлика како се проследени акредитивите, односно без разлика на типот на `Authentication` објектот. Единствениот апстрактен метод од филтерот е `attemptAuthentication(request, response)`. Преку овој метод, филтрите кои го наследуваат `AbstractAuthenticationProcessingFilter` потребно е да го изградат `Authentication` објектот од податоците во барањето и истиот да го валидираат.

На слика 6-16 е прикажан комплетниот процес на автентикација преку филтерот `UsernamePasswordAuthenticationFilter` изведен од овој апстрактен филтер:

- Филтерот креира `UsernamePasswordAuthenticationToken` со користење на параметрите од барањето за корисничко име и лозинка. Откако ќе се добие `Authentication` објектот од барањето, кој во овој момент не е валиден (`isAuthenticated()` враќа `false`), истиот се проследува на `AuthenticationManager` за да го провери.
- Резултатот од `AuthenticationManager` се враќа како резултат на методот `attemptAuthentication` од `UsernamePasswordAuthenticationFilter`. Доколку резултатот е `null`, тогаш се прекинува процесирањето на барањето и не се повикуваат наредните филтри. Ако акредитивите не се валидни или автентикацискиот објект не може да се валидира, `AuthenticationManager` фрла `AuthenticationException` исклучок. Во овој случај, најпрво се чисти `SecurityContextHolder` со повик на `SecurityContextHolder.clearContext()`. Потоа се повикува `RememberMeServices.loginFail` (доколку не е конфигуриран специфичен сервис, предефинирано се користи `NullRememberMeServices` имплементацијата, која не прави ништо при повик на нејзините методи).

На крај се активира конфигурираниот `AuthenticationFailureHandler`. Предефинираната имплементација `SimpleUrlAuthenticationFailureHandler` го пренасочува корисникот кон патеката која е конфигурирана со `http.formLogin().failureUrl("...")`. Кога нема конфигурирано патека за неуспешна најава, се враќа одговор со статус код 401 Unauthorized. Доколку има потреба од поразлично однесување при неуспешна најава, тоа може да се дефинира во посебна имплементација на `AuthenticationFailureHandler` и нејзино регистрирање со `http.formLogin().failureHandler(customFailureHandler)`.

3. Во случај кога автентикацијата е успешна, најпрво се известува конфигурираната `SessionAuthenticationStrategy` имплементација со повик на нејзиниот `onAuthentication` метод. Предефинираната стратегија е `NullAuthenticatedSessionStrategy`, која не превзема никаква акција. Оваа стратегија може да се конфигурира со повик на `http.sessionManagement().sessionAuthenticationStrategy(theStrategy)`. Потоа се поставува валидиранот `Authentication` објект во `SecurityContext` преку `SecurityContextHolder.getContext().setAuthentication(authResult)`. Наредно се повикува `rememberMeServices.loginSuccess` за да се запамети корисникот доколку е конфигуриран овој сервис. Потоа се објавува настан дека автентикацијата е успешна со повик на `ApplicationEventPublisher.publishEvent(InteractiveAuthenticationSuccessEvent)`. На крајот, се повикува конфигурираниот `AuthenticationSuccessHandler`. Предефинирано се користи `SavedRequestAwareAuthenticationSuccessHandler` имплементацијата која, доколку има зачувано барање во `RequestCache` (зачувано при неавтентициран обид за пристап), го проделдува за понатамошна обработка.. Ако нема такво зачувано барање или ако е поставено корисникот секогаш да се пренасочува на конфигурирана страна (`http.formLogin().defaultSuccessUrl("... true")`), тогаш корисникот ќе биде пренасочен на конфигурираната патека. Ако ваква предефинирана патека не е конфигуриран, тогаш корисникот се `SavedRequestAwareAuthenticationSuccessHandler` пренасочува на патеката `"/"`.



Слика 6-16: Обработка на барање со акредитиви за автентикација преку имплементација на `AbstractAuthenticationProcessingFilter`

Препораки за конфигурирање на автентикација?

Кога имаме нов тип на автентикација, треба да провериме дали некој од постоечките филтри ни се соодветни и дали може да го исполниме барањето со нивна конфигурација. Секогаш кога имаме корисничко име и лозинка како параметри во барањето, доволно е да креираме `UserDetailsService` кој ќе ги извлече деталите за корисникот. На овој начин ја искористуваме постоечката инфраструктура на филтри. Имплементација на наш `AuthenticationProvider` треба да направиме само ако имаме нов протокол за автентикација врз основа на податоци содржани во `Authentication` објектот. Ако дефинираме нов филтер за автентикација, препорачливо е тој да наследува од `AbstractAuthenticationProcessingFilter` и само да ги извлече податоците од барањето и да креира и валидира `Authentication` објект.

6.4.4 Авторизација со Spring Security

Сапо: да се додаде intro 1000-foot

Заштита на ниво на барање

Во претходното поглавје го објасниме процесот на автентикација со користење на Spring Security модулот. Џелиот процес на заштита на веб апликациите го спроведува филтерот `DelegatingFilterProxy` кој избира синџир со сигурносни филтри и го повикува. Во рамките на синџирот со сигурносни филтри се регистрирани филтрите за автентикација, кои при успешната автентикација поставуваат имплементација на `Authentication` интерфејсот во `SecurityContext`-от. Еден од последните филтри во синџирот со сигурносни филтри е `FilterSecurityInterceptor`, кој е задолжен за процесот на авторизација.

`FilterSecurityInterceptor` ги добива конфигурираните авторизациски правила за тековното барање од `SecurityMetadataSource`. Овде се добиваат дозволите за патеките кои се конфигурираат со `http.authorizedRequests().antMatchers(...)`. Делот `antMatchers` дефинира за кои барања ќе се применат дозволите кои се наредуваат со повиците на методите за конфигурирање на дозволи, како што се:

- `permitAll()` - дозволува пристап за сите
- `denyAll()` - не дозволува пристап на никој
- `authenticated()` - дозволува пристап само на автентицирани корисници (не се `anonymous`)
- `fullyAuthenticated()` - дозволува пристап на автентицираните корисници кои не се запаметени (не се добиени од `RememberMeService`)
- `anonymous()` - дозволува пристап само на неавтентицирани корисници
- `hasAuthority("...")` - дозволува пристап само на корисници кои во резултатите на методот `Authentication.getAuthorities()` ја содржат соодветната вредност
- `hasAnyAuthority(...)` - дозволува пристап само на корисници кои во резултатите на методот `Authentication.getAuthorities()` содржат некоја од наведените вредности
- `hasRole("...")` - дозволува пристап само на корисници кои во резултатите на методот `Authentication.getAuthorities()` ја содржат соодветната вредност со префикс `"ROLE_"`
- `hasAnyRole(...)` - дозволува пристап само на корисници кои во резултатите на методот `Authentication.getAuthorities()` содржат некоја од наведените вредности со префикс `"ROLE_"`
- `hasIpAddress("...")` - дозволува пристап само на барања направени од дадена IP адреса

- `rememberMe()` - дозволува пристап само на корисници кои се запаметени, односно добиени со `RememberMeService`
- `access("...")` - овозможува дефинирање на правила за пристап преку израз во јазикот SpEL (јазикот за изрази на Spring рамката ¹²). Сите претходно споменати методи за дефинирање на дозволи може да се дефинираат и преку изрази проследени како аргументи на `access("...")`. Во овие изрази може да се пристапи и до променливите `authentication`, `principal` и `request`, а со тоа и до нивните методи или својства кои можат да се искористат при одлуката дали ќе се дозволи пристап до ресурсот. Во овој метод може да комбинираат и повеќе изрази со користење на `or` и `and` логичките оператори: `"hasRole('admin')and hasIpAddress('192.168.1.0/24')"`. Дополнително, ако во контекстот на апликацијата е дефиниран бин со методи кои враќаат `boolean`, тие може да се повикаат со нивното име и аргументи така што пред името на бинот ќе се залепи `@Validate`: На пример, `"@securityBean.check(authentication,request)"` означува повик на метод `check` од бинот `securityBean` во апликацискиот контекст на Spring. Важно е да се напомене дека во изразите е овозможен пристап и до променливите од патеката кои се користат во дефинирањето на совпаѓањето на барањето. На пример, преку `serIdhttp.antMatchers("/user/u/**").access("@securityBean.checkUserId(authentication,#userId)")` се означува дека `#userId` од `access` методот ќе се однесува на променливата `{userId}` во патеката специфицирана во методот `antMatchers`.

Најголемиот дел од овие дозволи се однесуваат на автентицираниот корисник, односно `Authentication` објектот добиен со `SecurityContextHolder.getContext().getAuthentication()`. Како што може да се забележи од описот на методите за дефинирање на дозволите, улогите (roles) претставуваат специјална форма на авторитети (authorities). Предефинирано, улогите се авторитети кои се со префикс `"Role_"`. Притоа, кај методите `hasRole` и `hasAnyRole` треба да се изостави префиксот `"Role_"` кај вредностите што се проследуваат како аргумент.

`AccessDecisionManager` е компонентата која треба да донесе одлука дали тековното барање треба да биде авторизирано или не. Во случај кога не е дозволен пристап, се фрла исклучок `AccessDeniedException`, со кој потоа се справува `ExceptionTranslationFilter`-от. **Сашо:** да додадеме уште нешто, инаку делува како вишок

¹²<https://docs.spring.io/spring-framework/docs/5.3.x/reference/html/core.html#expressions>

Заштита на ниво на методи

Spring Security не нуди само авторизација која се справува со заштитата на HTTP барања, туку, дополнително нуди и заштита на повици на методи преку пресретнувачот `MethodSecurityInterceptor` кој што се имплементира преку концептот на аспектно ориентирано програмирање (AOP). Со AOP секој метод се енкапсулира во аспект. Пресретнувачот се повикува за секој од методите кои што се анонтираны со `@PreAuthorize`, `@PostAuthorize`, `@PreFilter`, `@PostFilter` или `@Secured` при што се пренесуваат информациите за најавениот корисник и повиканиот метод. Според овие информации и според правилата наведени во анотациите, `MethodSecurityInterceptor` проверува дали корисникот има привилегија да го повика методот. Доколку е дозволен пристапот до методот, тој се повикува, а доколку не е дозволен пристап, се фрла соодверна грешка.

Spring Security дополнително нуди и заштита и на каналите за размена на пораки (`MessageChannel`) преку `ChannelSecurityInterceptor` пресретнувачот. Во овој случај, се пресретнува секоја порака, при што се прави повик на `ChannelSecurityInterceptor`, кој со помош на конфигурациите за таа порака одлучува дали смее или не смее да се продолжи со справувањето со соодветната порака.

За да се овозможат анотациите за заштита на методи, потребно е на конфигурацијата за безбедноста на апликацијата да се постави анотација: `@EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true)`. `@Secured` анотацијата единствено може да означи за кои доделени авторитети (`GrantedAuthority`) на тековниот корисник може да се повика анотираниот метод. Како што е прикажано во првите два методи од изворниот код [6.11](#), мора да се користат целосните имиња на доделените авторитети, а дополнително може и да се означи дека се дозволува пристап само за неавтентицирани корисиници со `IS_AUTHENTICATED_ANONYMOUSLY`.

Во рамките на `@PreAuthorize`, `@PostAuthorize`, `@PreFilter` и `@PostFilter` може да се користат изразите кои може да се искористат и во `access("...")` конфигурацијата на дозволите. Дополнително, во изразите може да се референцираат и аргументите од методите анотирани со `@P` со `#` како префикс пред името на променливата, како што е прикажано во `doSomething` методот од изворниот код [6.11](#). Исто така, може да се референцира и објектот што се вараќа како резултат во `@PostAuthorize` со променливата `returnObject`. За филтрирање на колекции може да се користат `@PreFilter` и `@PostFilter`, каде се наведува услов за секој од елементите на колекцијата претставен со имплицитната променлива `filterObject`.

```

2  @Secured("ROLE_TELLER")
3  public Account post(Account account, double amount){ ... }
4
5  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
6  public Account readAccount(Long id) { ... }
7
8  @PreAuthorize("hasRole('USER')")
9  @PostAuthorize("returnObject.owner == authentication.name")
10 Account readAccount(Long id); { ... }
11
12 @PreAuthorize("#c.name == authentication.name")
13 public void doSomething(@P("c") Contact contact) { ... }
14
15 @PreAuthorize("hasRole('USER')")
16 @PostFilter("filterObject.name == authentication.name")
17 public List<Contact> getAll() { ... }

```

Изворен код 6.11: Анотации за конфигурирање на авторизациски пристап до методи

Само: да се дообјасни кодот

TODO: Објаснување на кодот подолу

```

1 import static
2     org.springframework.security.test.web.servlet.response.SecurityMockMvcResultMatchers.*
3 import static
4     org.springframework.security.test.web.servlet.setup.SecurityMockMvcConfigurers.*;
5
6 @ExtendWith(SpringExtension.class)
7 @ContextConfiguration(classes = SecurityConfig.class)
8 @WebAppConfiguration
9 public class ShowcaseTests {
10
11     @Autowired
12     private WebApplicationContext context;
13     private MockMvc mvc;
14
15     @BeforeEach
16     public void setup() {
17         mvc =
18             MockMvcBuilders.webAppContextSetup(context).apply(springSecurity()).build();
19     }

```

```

17
18     @Test
19     @WithMockUser(username="admin", roles={"USER", "ADMIN"})
20     public void getMessageWithMockUserCustomUser() {
21         String message = messageService.getMessage();
22         // do assertion
23     }
24
25     @Test
26     @WithAnonymousUser
27     public void anonymous() throws Exception {
28         String message = messageService.getMessage();
29     }
30
31     @Test
32     public void test() {
33         mvc.perform(formLogin("/auth").user("u", "admin").password("p", "pass"))
34             .andExpect(unauthenticated());
35         mvc.perform(get("/admin").with(user("admin").password("pass").roles("USER", "ADMIN")))
36             .andExpect(status(200));
37     }
38 }
```

Изворен код 6.12: ShowcaseTests

6.4.5 Преглед на Spring Security функционалности

Spring Security е рамка која обезбедува автентикација, авторизација и заштита од вообичаени напади. Со овој модул се овозможува конфигурабилна заштита на сите влезни точки во апликацијата, како што се пресретнување на барањата за веб апликациите и пресретнување на повиците на методите. При овие пресретнувања, се уважуваат конфигурираните правила за авторизација.

Дополнително, за веб апликациите се нудат предефинирани филтри кои поддржуваат различни протоколи за автентикација и не заштитуваат од некои од најчестите напади. Сепак, се овоможуваат повеќе точки на проширување на овој модул, при што може да се дефинираат специфични сервиси, овозможувачи за автентикација и филтри со кои може да се имплементира уште понапредна заштита, во зависност од потребите.

Накратко кажано, Spring Security модулот претставува де-факто стандард за заштита на Spring апликациите.

Додаток А

Полиња на HTTP заглавје

Листата на HTTP заглавја кои се сретнуваат кај барањето или одговорот е прилично долга и може да се погледне во официјалниот RFC документ за протоколот¹. Сепак, како поважни полиња од аспект на веб програмирање, можат да се издвојат следните:

- **Host** преставува неизбежен дел од барањето и го содржи делот од побараната URL адреса кој го означува името на домаќинот, односно серверот од каде што се бара ресурсот. Адресата на самиот ресурс е сместена во почетната линија на барањето. На пример, ако во прелистувачот се внесе URL адресата `http://finki.ukim.mk/springbook/course/intro/index.html`, тогаш полето host ќе има вредност `http://finki.ukim.mk`, а полето за URI на ресурсот во првата линија на бараќето ќе има вредност `/springbook/course/intro/index.html`. Ако на истиот сервер се хостираат повеќе сайтови, тогаш барањето до секој од нив ќе има иста вредност на host, но ќе се разликува по вредноста на полето за URI.
- **Referer** ја означува адресата на страната од која потекнува барањето. На пример, ако на страната `http://finki.ukim.mk/springbook/courselist.html` кликнеме на линк до `http://finki.ukim.mk/springbook/course/intro/index.html`, тогаш ќе се генерира барање кон овој ресурс, но страната која го генерирала тоа барање, односно Referer е `http://finki.ukim.mk/springbook/courselist.html`.
- **User-Agent** носи информации за типот на апликацијата, оперативниот систем, производителот на софтверот и слично за хостот кој го генерира барањето. Од овие информации може да се види дали барањето потекнува од мобилен уред или од десктоп компјутер, оперативниот систем на корисни-

¹Комплетна листа на полиња од HTTP заглавје: <https://www.rfc-editor.org/rfc/rfc9110.html>

кот како и прелистувачот кој го користи.

- **Server** содржи информации за серверот кој го опслужил барањето, односно конкретниот софвер кој има улога на веб сервер
 - **Location** поле карактеристично за одговорот кое ја содржи URL адресата на ресурсот во случај на пренасочување. На пример, ако статусниот код на одговорот е 301, тогаш серверот мора да ја специфицира новата URL адреса на ресурсот кон која прелистувачот треба да генерира ново барање.
 - **Content-Type** го дефинира типот на содржина кој се пренесува во телото на пораката. Кога во телото се пренесуваат податоци, тие преставуваат низа од бајти па серверот или клиентот кој ја обработува таа порака нема информација за каков тип на податоци станува збор. Оваа дилема ја разрешува полето **Content-Type** кое точно го дефинира типот на податоци. Ако станува збор за одговор, типот му дава до знаење на веб прелистувачот за типот на содржина во телото со цел да знае како да ја обработи. На пример ако станува збор за html содржина, тогаш ќе ја испарсира и ќе ја прикаже на еранот. Ако станува збор за .zip датотека тогаш ќе ја зачува на предефинирата патека како датотека. Ова поле се користи и во HTTP барањата со цел да му се даде до знаење на серверот како да ги обработи податоците во телото кои се испратени до него преку барањето. На пример, ако станува збор за податоци внесени од некоја форма, ќе ги искористи за да ги прочита и обработи. Ако станува збор за некоја датотека (при прикачување на датотека), тогаш потокот од податоци ќе го зачува како датотека на некоја предефинирана локација. Во случај да не се практикат податоци преку телото на пораката, ова поле е изоставено од заглавјето. За типовите на содржини се користи конвенција на мултимедиски типови наречена MIME (Multipurpose Internet Mail Extensions) која првично се користела за дефинирање на типовите податоци во електронските пораки, но набрзо станува прифатена кај сите мрежни протоколи кои работат со различни типови на податоци. Најчесто се сокоти од тип и подтип одвоени со коса црта (/). Комплетната листа на MIME типови може да се најде на страницата на IANA². Некои од почесто користените типови и нивните интерпретации се:
- **text/html** - .html датотека
 - **text/javascript** - .js датотека
 - **text/css** - .css датотека
 - **text/plain** - .txt текстуална датотека
 - **image/jpeg** - .jpeg или .jpg слика
 - **image/png** - .png слика

²Комплетна листа на MIME типови: <https://www.iana.org/assignments/media-types/media-types.xhtml#application>

- `video/mp4` - .mp4 видео (видео+аудио во mp4 пакет)
 - `audio/aac` - .aac аудио датотека
 - `application/json` - .json датотека со податоци во JSON формат
 - `application/pdf` - .pdf датотека
 - `application/zip` - .zip датотека
 - `application/vnd.ms-powerpoint` - .ppt датотека
 - `application/x-www-form-urlencoded` - податоци од html форма кои се пренесуваат до сервер во POST барања. Ако во некоја форма се внесуват податоци во компоненти од типот `<input>` (освен од типот `file`), на пример текстуално поле, листа од опции и слично и тие се испратат до серверот (се притиска копче), податоците се пренесуваат во телото на пораката во формат на парови клуч=вредност одвоени со & според енкодирањето кои се користи за URL адреси (сите знаци кои не се алфанимерички се кодираат со %HH, на пример, '=' се кодира со '%20')
 - `multipart/form-data` - се користи за прикачување (анг. upload) на една или повеќе датотеки на сервер преку html компонента од типот `<input type="file">` или за комбиниран пренос на податоци од форма вклучувајќи и прикачување на датотеки.
- `Content-Length` ја дава големината на податоците во телото изразено во байти. Приемникот на пораката ја користи оваа информација за да знае колку байти да исчита после празната линија. податоци да обработи
 - `Content-Encoding` ако содржината на телото е компресирана со цел да се намали пренесениот сообраќај во мрежата, го дефинира кодекот за компресија. Ако станува збор за одговор, тогаш прелистувајќи го користи кодекот за да ја декомпресира содржината и потоа да ја обработи во зависност од типот во `Content-Type`. Вредноста на ова поле може да биде gzip, compress, deflate, br кои означуваат различни алгоритми за компресија или identity за да се означи дека содржината не е компресирана бидејќи веќе е компресирана со нејзиниот формат (пр. jpg е компресирана слика која не може понатаму да се компресира) или серверот е веќе доволно оптоварен и не може да троши дополнителни ресурси за компресија.
 - `Content-Language` го означува јазикот на корисниците за кои е наменета содржината во телото на пораката. Ако на серверот има иста содржина на повеќе јазици, клиентот и серверот преговараат за јазикот кој клиентот го преферира преку полето `Accept-Language`. Во зависност од одлуката, серверот го означува јазикот документот кој решил да го прати како содржина на телото. Ова поле не секогаш значи дека јазикот на содржината е ист како и означените јазици. Јазиците се означуваат со нивниот интернационален

код од два знака. На пример en и mk се кодови за английски и македонски јазик, соодветно. Ако јазикот се збори во повеќе земји, а со тоа има одредени разлики во зависност од земјата, тогаш се наведува интернационалниот код на земјата на кој се прилепува пртичка (-) и интернационалниот код на јазикот. На пример со кодовите en-us и en-uk се означува английски јазик кој се зборува во САД и Обединетото Кралство, соодветно.

- **Accept** содржи еден или повеќе MIME типови кои ги поддржува прелистувачот и се испраќа на почетокот во процесот на преговарање со серверот. Според оваа листа серверот знае кој тип на содржини може да му ги испраќа на прелистувачот. На пример ако прелистувачот поддржува само html содржина, тогаш неговата вредност е `text/html`. Ако поддржува било каков тип на текстуални содржини вредноста е `text/*`. Ако поддржува само html и слики тогаш вредноста е `text/html, image/*`. Најголемиот број на прелистувачи ги поддржуваат сите MIME типови, па предефинирана вредност на ова поле е `*/*`. Во случај да е потребно да се даде тежински фактор на одреден тип кој ја означува преференцата на прелистувачот за одреде тип тогаш се додава `;q=` и тежината во опсег (0,1). На пример `text/html, text/css, image/*;q=0.9, */*;q=0.8` означува дека прелистувачот подеднакво ги поддржува `text/html, text/css` со максимална преференца ($q=1$, максимална предефинирана вредност, но не се запишува), па потоа сликите ($q=0.9$, помала тежина) и на крај сите останати содржини ($q=0.8$, најмала тежина). Овој тежински начин на преференци се употребува секогаш кога има листа од повеќе опции.
- **Accept-Language** содржи еден или повеќе јазици на содржина кои ги очекува прелистувачот од серверот. Се користи во процесот на преговарање за јазик со серверот кои прелистувачот ги испраќа во барањето. Јазиците се конфигурирани или во оперативниот систем или во самиот прелистувач. Ако серверот располага со повеќе верзии на содржината во повеќе јазици, тогаш одбира јазик преку посебен процес на преговарање (се зема предвид вредност на q , но одлуката зависи од серверот) и го известува прелистувачот за избраниот јазик преку полето **Content-Language** во одговорот. Ако серверот ги нема јазиците кои ги преферира прелистувачот, тогаш го испраќа предефинираниот јазик (поставен во неговата конфигурација). Ако серверот има само еден јазик на содржина, тогаш ја испраќа таа содржина без разлика каква е преференцата на прелистувачот.
- **Accept-Encoding** - содржи еден или повеќе алгоритми за компресија на содржината на телото кои ги поддржува прелистувачот. Ако прелистувачот не поддржува компресија, ова поле е празно, па серверот испраќа податоци во одговорите кои не се компресирани. Ако и прелистувачот и серверот под-

држуваат повеќе кодеци (од кои барем еден заеднички) тогаш серверот го компресира телото на одговорот и го известува прелистувачот за избраниот кодек преку полето **Content-Encoding**.

- **Set-Cookie** - содржи парови на име-вредност со колачиња кои ги испраќа серверот во одговорот за да прелистувачот ги зачува во својата постојана меморија и да ги испраќа со секое следно барање
- **Cookie** - содржи парови име-вредност со колачиња кои ги испраќа прелистувачот до серверот во барањето
- **Connection** - означува дали хостот кој ја испраќа пораката сака TCP конекцијааста да се држи отворена (има вредност `keep-alive`) или да се затвори откако ќе се добие бараниот ресурс (има вредност `close`). Ако и серверот и клиентот се согласат да ја држат отворена, тогаш сите барања се пренесуваат преку истата конекција. Кај верзија HTTP/2 не се користи бидејќи преносот на податоците се одвива преку повеќе паралелни TCP конекции.
- **Keep-Alive** - означува колку време да се држи врската отворена (ако двете страни се согласат да ја држуваат отворена). Како вредност се специфицира параметарот `timeout` кој означува колку време да се држи отворена без да се испраќаат податоци (анг. `idle`) и параметарот `max` кој означува колку барања може да се испратат пред да се затвори.
- **Cache-Control** - го има и во барањето и во одговорот и дефинира правила кои треба да ги запазат сите страни во однос на испраќање кеширани содржини од страна а серверот, клиентот или прокси сервери преку кои се одвива комуникацијата
- **Last-Modified** - ги содржи датумот и времето кога ресурсот бил последен пат модифициран од страната на серверот
- **ETag** - содржи уникатен идентификатор на секоја верзија ја ресурсот и најчесто е хаш вредност добиена од самата содржина. Се користи за споредба на верзии за кеширање
- **If-Match** - содржи вредност на ETag на баран ресурс и барањето се исполнува само ако ETag на ресурсот на серверот се совпаѓа со неговата вредност
- **If-Modified-Since** - содржи датум и време. Ако ресурсот е променет после тој датум и време, тогаш серверот враќа статусен код 200 и го испраќа ресурсот на клиентот. Во спротивно, серверот враќа одговор без содржина во телото со статусен код 304, а клиентот ја искористува претходно кешираната верзија.
- **Authorization** - содржи информации за најава кои се користат за автентификација на корисникот со серверот, овозможувајќи пристап до заштитетни ресурси. Го користат голем број на шеми за автентификација.
- **WWW-Authenticate** - содржи автентикациски методи поддржани од серверот

кои му се испраќаат на клиентот откако ќе се обиде да пристапи до заштичен ресурс (во одговор со статус 401)

Work in progress

Додаток Б

Објекти за HTTP барање и одговор

Објектите кои ги претставуваат HTTP одговорите и барањата кои ги обработуваат сервлетите се инстанци од класите `HttpServletRequest` и `HttpServletResponse` кои се дел од пакетот `javax.servlet.http`. Всушност, податоците кои ги разменува клиентот со серверот и серверот со клиентот се содржани токму во овие објекти, па затоа е важно да се познаваат методите кои овозможуваат интеракција со нив и нивно уредување.

Б.1 HttpServletRequest

Класата `HttpServletRequest` содржи елементи од HTTP барањето кои се однесуваат на адресата на барањето, параметрите кои се пренесуваат во адресата, заглавјата и телото на барањето. За да се пристапи до елементите поврзани со различни делови од оригиналната URL адреса се користат методите:

- `String getRequestURI()` - ја враќа адресата од првата линија на HTTP на барањето (без полето Host) од почетокот до низата на пребарувањето (анг. query string). Не ги вклучува протоколот, домаќинот и портата ниту низата на параметри.
- `StringBuffer getRequestURL()` - ја враќа адресата од почетокот до низата на пребарувањето (анг. query string). Ги вклучува протоколот, домаќинот и портата, но не и низата на параметри
- `String getContextPath()` - ја враќа адресата на контекстот, односно делот од адресата во кој е мапирано името на веб апликацијата
- `String getServletPath()` - ја враќа адресата на сервлетот, односно делот од адресата во кој е мапирано името на сервлетот
- `String getPathInfo()` - го враќа последниот дел од адресата кој следи после името на сервлетот

- `String getQueryString()` - ја враќа низата на параметри кои се пренесуваат преку URL на барањето

За да видиме како функционираат овие методи во пракса, ќе го разгледаме примерот со следната URL адреса:

GET `http://localhost:80/SpringBook/course/intro/index.html?content=1&page=2`

За секој од методите ги добиваме следните вредности

```
String uri = request.getRequestURI()
    ↵ //uri="/SpringBook/course/intro/index.html"
String url = request.getRequestURL()
    ↵ //url="http://localhost:80/SpringBook/course/intro/index.html"
String context = request.getContextPath() //context="/SpringBook"
String servlet = request.getServletPath() //servlet="/course"
String path = request.getPathInfo() //path="intro/index.html"
String query = request.getQueryString() //query="content=1&page=2"
```

За пристап до заглавјата и останати елементи од барањето се користат методите

- `String getMethod()` - го враќа типот на HTTP барање (пр. GET)
- `String getRemoteAddr()` - ја враќа IP адресата на клиентот кој го испратил барањето
- `String getHeader(String header)` - ја враќа вредноста на заглавјето со име `header`
- `Enumeration<String> getHeaders(String header)` - ги враќа сите вредности на заглавјето со име `header` како енумерација. Се користи за заглавја кои содржат листа од вредности одвоени со запирка (пр. заглавјето Accept-Language може да има вредност en-US,fr-CA).
- `Enumeration<String> getHeaderNames()` - ги враќа имињата на сите заглавја во барањето
- `String getContentType()` - го враќа типот на содржина на податоците во телото на барањето
- `int getLength()` - ја враќа големината во байти на телото на барањето
- `Cookie[] getCookies()` - враќа листа од колачиња

Преку барањето можат да се пренесуваат и кориснички дефинирани параметри во низата на пребарувањето (анг. query string) или во телото на барањето. Едено барање може да содржи параметри со исто име и во низата на параметри и на телото. За пристап до нив се користат следните методи:

- `String getParameter(String name)` - ја враќа вредноста на параметарот со име `name`. Вредноста зависи од типот на методот, па ако станува збор за GET, го враќа параметарот од низата на параметри во барањето, а ако станува збор за POST го враќа параметарот од телото на барањето. Ако параметарот не постои, се враќа `null`
- `Enumeration<String> getParameterNames()` - ги враќа имињата на сите параметри во барањето и во низата на параметри од адресата и од телото
- `String[] getParameterValues(String name)` - ја враќа вредноста на сите параметри со име `name` во барањето, почнувајќи од низата на параметри од адресата, па потоа и од телото

При креирање на POST барања од страна на HTML форми, содржината во телото на барањето да не од типот *application/x-www-form-urlencoded*. Во овој случај сервлет контејнерот ги третира испратените податоци како стандардни параметри и знае како да ги препознае и врати преку `getParameter` методот. Меѓутоа, доколку содржината на телото е во друг формат, податоците од телото на барањето може да се пристапат преку влезни потоци од байти или знаци кои овозможуваат да се исчитуваат податоци од оригиналното HTTP барање преку методите:

- `ServletInputStream getInputStream()` - враќа поток од байти од кој може да се читаат податоците од телото байт по байт.
- `BufferedReader getReader()` - враќа поток од знаци од кој може да се читаат податоците од телото знак по знак.

Телото на пораката е поток од податоци кои пристигнуваат од клиентот. Потокот се отвора, се читаат податоците по редослед и на крајот се затвора. Во овој процес, податоците можат да се изминат само еднаш (потоа податоците од потокот се ослободуваат од меморијата) без разлика дали објектот ќе се проследува до други сервлети. Штом податоците еднаш се изминат, нема да може да се пристапи ни до параметрите од телото користејќи го методот `String getParameter(String name)` бидејќи самите параметри биле веќе еднаш изминати.

Постојат и други важни методи од оваа класа за управување со корисничка сесија и атрибути, кои ќе бидат представени во продолжение на оваа глава.

B.2 HttpServletResponse

Класата `HttpServletResponse` содржи елементи од HTTP одговорот кој ќе се испрати до клиентот. Објектот е иницијално го креира сервлет контејнерот и поставува вредности на заглавјата, но негова содржина, особено телото, се пополнува во методите за обработка на барањата во сервлетите. За разлика од `HttpServletRequest`

каде методите се однесуваа на читање на дојдовните податоци, оваа класа содржи и методи за поставување вредности кои ќе се испратат до клиентите, вклучувајќи го телото на одговорот чија содржина се прикажува во прелистувачот.

Поважни методи од оваа класа се:

- `public void setHeader(String header, String value)` - ја поставува вредноста на заглавјето со име `header` на вредност `value`. Ако заглаје со наведеното име веќе постои во одговорот, тогаш старата вредност се пребришува со новата вредност. Во спротивно, заглавјето се додава на крајот од листата на заглавја.
- `void addHeader(String header, String value)` - Се додава ново заглавје со име `header` и вредност `value` на крајот од листата на заглавја
- `boolean containsHeader(String header)` - проверува дали веќе е поставено заглавје со име `header`
- `String getHeader(String header)` - ја враќа вредноста на заглавјето со име `header`
- `Enumeration getHeaders(String header)` - ги враќа сите вредности на заглавјето со име `header` како енумерација
- `Enumeration getHeaderNames()` - ги враќа имињата на сите заглавја во одговорот
- `void setContentType(String type)` - го поставува типот на содржина на податоците во телото на одговорот според кој прелистувачот одлучува како да ги прикаже податоците (пр. "text/html" за html содржина). Се повикува пред да почне да се запишува во телото.
- `void setStatus(int status)` - Поставува код за статусот на одговорот (пр. 200 за успешен одговор)
- `int getStatus(int status)` - го враќа поставениот статусот на одговорот
- `void sendError(int status, String message)` - Испраќа одговор за грешка со статус `status` и порака `message`
- `void addCookie(Cookie cookie)` - додава колаче во листата на колачиња

За да се пристапи до телот на HTTP одговорот со цел во него да се запишат податоците за приказ кај корисникот се користи излезен поток на байти или знаци. За да се добие пристап до овие потоци се користат методите

- `ServletOutputStream getOutputStream()` - враќа поток од байти во кој се запишуваат податоците од телото байт по байт. Се користат методите `print()` и `println()` за да се запише знак, односно линија во потокот
- `PrintWriter getWriter()` - враќа поток од знаци во кој се запишуваат податоците од телото знак по знак (пр. запишување на html код)

И при работа со излезните потоци за телото на одговорот важи правилото дека податоците се запишуваат во секвенцијален редослед и откако потокот ќе се затвори, не може повторно да се отвори за да се допишуваат нови податоци бидејќи со самото затворање всушност се испраќа одговорот до клиентот. **TODO:** Дали да се даде пример или да се искористи примерот за имплементација и да се вметнат методи од барањето

Б.3 Правила за повеќекратни пресликувања на сервлети

Поради можноста за дефинирање на URL адреси со регуларни изрази, можни се ситуации каде една URL адреса може да се пресликува во повеќе сервлети. За да се елиминираат било какви двосмислености, при пресликувањето на дојдовните URL адреси, сервлет контејнерот се води по следниве правила:

1. Се преферира егзактно покlopување наспроти покlopување со регуларен израз.
2. Се преферираат подолгите покlopувања.
3. Ако не се пронајде ниту едно покlopување, се прави предефинирано пресликување на сервлетот одговорен за коренот на апликацијата /.

На пример, ако за една апликација се дефинирани само две пресликувања `"/products/*"` и `"/products/software/*"`, и ако пристигне URL адреса `"/products/software?name=spring"`, тогаш, иако адресата се преклопува со двете пресликувања, ќе биде одбран сервлетот кој одговара на второто пресликување бидејќи тоа е подолго. Ако пристигне URL адреса `"/e-books"`, нема да да има совпаѓање ниту со едно од пресликувањата, па затоа ќе биде одбран сервлетот кој одговара на пресликувањето `"/"`. Во нашиот пример, ниту еден сервлет не е доделен за ова пресликување, па затоа апликацијата ќе исфрли грешка. За да се одбегне таков тип на грешки, потребно е во конфигурациската датотека секогаш да има сервлет кој ќе одговара на пресликувањето `"/"`, односно предефиниран сервлет (анг. default servlet).

Б.4 Дефиниција на параметри на сервлети

Во делот за дефиниција на сервлетите може да се постават кориснички дефинирани иницијални параметри кои ќе се однесуваат на специфичен сервлет. Овие параметри ги поставува програмерот, а до нивната вредност може да се пристапи преку кодот на сервлетот на кој се однесуваат. Вредности на параметрите се

поставуваат еднаш и не можат да се менуваат од страна на сервлетот во текот на извршувањето на апликацијата (READ ONLY). Параметрите се дефинираат како парови од елементи `<param-name>`, `<param-value>` сместени помеѓу ознаките `<init-param>...</init-param>`. За секој сервлет може да се дефинира произволен број на параметри.

Во дадениот коден сегмент се дефинираат два параметри за сервлетот `login`. Првиот параметар е со име `login-attempts` и вредност `3`, а вториот е со име `lock-account` и вредност `true`.

Listing 2 Дефиниција на параметри на servlet

```
<servlet>
    <servlet-name>login</servlet-name>
    <servlet-class>mk.ukim.finki.Login</servlet-class>
    <init-param>
        <param-name>login-attempts</param-name>
        <param-value>3</param-value>
    </init-param>
    <init-param>
        <param-name>lock-account</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>
```

Исто како дополнителниот начин за конфигуирање на сервлетите и пресликувањата со помош на анотацијата `@WebServlet`, така и нивните параметри можат да се дефинираат како листа од анотирани параметри `@WebInitParam` со својства `name` и `value` која се дава како вредност на параметарот `initParams` од анотацијата `@WebServlet`. За примерот од сликата 2 параметрите на сервлетот `login` би се дефинирале со анотирање на класата `Login` на следниот начин:

```
@WebServlet(name = "login", value = "/login", initParams = {
    @WebInitParam(name = "login-attempts", value = "3"),
    @WebInitParam(name = "lock-account", value = true)
})
public class Login extends HttpServlet {
    ...
}
```

Б.5 Конфигурација на редослед на вчитување на сервлети

Предефинирано однесување на контејнерот е да ги чита класите за сервлетите кога ќе пристигне првото барање за секоја од нив. Со ваквиот пристап, обработката на првите барања за сервлети ќе траат подолго време од вообичаено. За да се елиминира ова доцнење, може да се постави сервлет контејнерот да ги прочита класите за секој од сервлетите при самото вклучување на апликацијата. Оваа поставка се прави во конфигурациската датотека `web.xml`, каде за секој сервлет во елементот за негова дефиниција `<servlet>` се додава елемент `<load-on-startup>` со целоброяна вредност која го означува редоследот на читање на класата за тој сервлет. Сервлети со помали вредности ќе бидат посекоро прочитани од сервлети со поголеми вредности. Сите сервлети со позитивни вредности на овој параметар ќе бидат прочитани веднаш по подигањето на апликацијата, но во различен редослед. За негативни вредности, сервлет контејнерот самиот одлучува кога ќе ја прочита класата за сервлетот. Ако параметарот е изоставен во дефиницијата за сервлет, сервлетот се чита кога ќе пристигне првото барање за него.

Во дадениот пример, веднаш при вклучувањето на апликацијата се вчитува сервлетот `login`, а по него и сервлетот `cart`, без да се чека да пристигне барање за било кој од нив.

Listing 3 Дефиниција на редослед на вчитување на сервлет

```
<servlet>
    <servlet-name>login</servlet-name>
    <servlet-class>mk.ukim.finki.Login</servlet-class>
    <load-on-startup>0</param-name>
</servlet>
<servlet>
    <servlet-name>cart</servlet-name>
    <servlet-class>mk.ukim.finki.ShopingCart</servlet-class>
    <load-on-startup>2</param-name>
</servlet>
```

Б.6 Контекстни настани и слушачи на настани

При користење на контекстни атрибути, често е потребно тие да се иницијализираат со почетни вредности и истите да се избришат кога апликацијата ќе заврши. Поради нивниот глобален опсег на важење на ниво на целата апликација, важно

е вредностите да се иницијализираат пред да бидат креирани сервлетите и да бидат избришани откако ќе се избришат сите сервлети. Но кои се моментите кога тоа може да се постигне и на кој начин? Кога има промени во состојбата контекстот, а тоа се негово креирање или бришање, контејнерот еmitува соодветен настан кој е инстанца од типот `SerlvetContenxtEvent`. Оваа класа има конструктор во кој се пренесува инстанца од контекстот

```
ServletContextEvent(ServletContext e)
```

до која може да се пристапи преку единствениот метод дефиниран за класата `SerlvetContenxtEvent`, а тоа е

```
public ServletContext getServletContext()
```

Методот е корисен за да се добие референца до контекстот за преку неа да иницијализираат или избришат контекстните атрибути при обработка на соодветниот настан.

За да може се фати и обработи настанот, потребно е да се дефинира класа која ќе има улога на слушач на настани и за секој настан ќе дефинира метод кој ќе го опслужува кој ќе се повикува во моментот кога настанот ќе се случи. За да се воведе оваа функционалност, класата со улога на слушач на контекстни настани мора да го имплементира интерфејсот `ServletContextLlistener` кој ги дефинира следните два методи:

```
public void contextInitialized(ServletContextEvent e)  
public void contextDestroyed(ServletContextEvent e)
```

Првиот методот се повикува веднаш откако ќе се иницијализира контекстот. Во овој момент сите контекстни параметри се веќе прочитани од `web.config` и нивните референци се сместени во контекстот. Настанот се користи за да се креираат атрибути и да се постават нивните почетни вредности или пак за да се иницијализираат компоненти кои ќе ги користат сите сервлети како што е на пример конекција до база на податоци. Вториот метод се повикува пред контекстот да се избрише и се користи за да се зачуваат атрибутите во постојано складиште (на пример, во база) пред да се избришат или за да се ослободат ресурси на компоненти кои ги користат сите сервлети како што е на пример затворање на конекција до база на податоци. И двата метода како аргумент го пренесуваат настанот дефиниран со класата `SerlvetContenxtEvent` преку кој се овозможува пристап до контекстот, а со тоа и до неговите параметри и аргументи.

Во примерот за контекстниот атрибут `cartVisits` кој води евидентија за сите посети на потрошувачката кошника во кодниот исечок 2.13, нагласивме дека неговата иницијализација во `init()` на сервлетот не е најсоодветно место бидејќи

ако се пристапи до атрибутот од друг серверт, пред да се повика `init()` на сервертот `cart`, тој атрибут нема да посоти. Сега, откако ги воведовме слушачите на настани како место за обработка на настани поврзани со промени на контекстот, ќе го надградиме примерот така што, наместо да го иницијализираме атрибутот `cartVisits` во `init()` на сервертот, ќе дефинираме класа слушач на настани поврзани со промена на контекстот дадена во кодниот исечок 4.

Listing 4 Пример за иницијализација на контекстни атрибути во слушач на контекстни настани

```
public class EshopListener implements ServletContextListener {  
    public void contextInitialized(ServletContextEvent e)  
    {  
        // initialize connection to database  
        // get number of saved visits from database and store it in visits  
        ServletContext context = e.getServletContext();  
        context.setAttribute("cartVisits", visits);  
    }  
    public void contextDestroyed(ServletContextEvent e) {  
        ServletContext context = getServletContext();  
        Integer visits = (Integer) context.getAttribute("cartVisits");  
        //store visits in database  
        //close connection to database  
    }  
}
```

Во примерот, во методот кој се повикува откако ќе се иницијализира контекстот `contextInitialized()` се иницијалира конекција кон база и се чита бројот на претходно зачувани посети откако апликацијата била претходно исклучена (имплементацијата е надвор од целите на оваа глава). Потоа, се креира контекстен атрибут чија вредност се иницијализира на прочитаниот број на посети. Во текот на животнот век на апликацијата вредноста на атрибутот се инкрементира во сервертите кои се дел од неа. Откако апликацијата ќе се исклучи и ќе се избрише контекстот, се повикува методот `contextDestroyed()` во кој се зачувува последната вредност на атрибутот во база на податоци со цел при следното вклучување на апликацијата да биде прочитана од база и поставена како почетна вредност на контекстионт атрибут `cartVisits`.

За контејнерот да направи инстанца од слушачот на настани при вклучување на апликацијата, потребно е тој да се конфигурира во `web.config` така што ќе се додаде името на класата која го имплементира слушачот во јазолот `<listener>` (директно дете на `<web-app>`) помеѓу ознаките `<listener-class>...<listener-class>`. Конфигурацијата за нашиот пример со е-продавница е дадена во кодниот исечок 5.

Listing 5 Конфигурација на `web.config` за дефиниција на слушач на котенсктни настани

```
<web-app>
    ...
    <listener>
        <listener-class>EshopListener</listener-class>
    </listener>
    ...
</web-app>
```

Друг, поедноставен начин за конфигурација на слушачот на настани е во кодот на неговата имплементација со користање на анотацијата `@WebListener` пред дефиниција на класата. Ако се користи анотација, тогаш не треба да се прават поставки во `web.config`.

Покрај настани кои се испалуваат кога ќе настане промена на состојбата на контекстот, постојат и настани кои се поврзани со промената на состојбата на атрибутите, односно нивно креирање, менување и бришење. Во тој случај, настаните се претставени со класата `ServletContextAttributeEvent` со конструктор во кој се пренесува инстанца од контекстот, името на променитиот атрибут и неговата вредност:

```
public ServletContextAttributeEvent(ServletContext source, String name,
    → Object value)
```

До името и вредноста на променетиот атрибут се пристапува преку нејзините методи

```
public String getName()
public Object getValue()
```

За да се опслужат настаниите при промена на состојбата на атрибути, потребно е да се дефинира слушач на настани кој ќе го имплементира интерфејсот `ServletContextAttributeListener` кој ги дефинира методите:

```
public void attributeAdded(ServletContextAttributeEvent e)
public void attributeRemoved(ServletContextAttributeEvent e)
public void attributeReplaced(ServletContextAttributeEvent e)
```

Првиот метод се повикува кога се додава нов атрибут во контекстот, вториот кога се отстранува атрибут од контекстот а третиот кога вредноста на постоечки атрибут ќе се замени со нова вредност. За да се пристапи до името и вредноста на соодветниот атрибут се користат методите на објектот кој од типот `ServletContextAttributeEvent` кој се пренесува како нивен аргумент.

За да се вклучи слушачот на настани за промена на атрибути на контекст, името на класата треба да се додаде помеѓу `<listener-class>...<listener-class>` како нов јазол `<listener>` во `web.xml`.

Б.7 Настани и слушачи на настани за барање

Како и кај контекстот, така и кај барањето се емитуваат настани при промена на неговата состојба. Настаните кои се важен дел од животниот циклус на едно барање се негова иницијализација од страна на контејнерот во моментот кога ќе пристигне и пред да биде доделено на сервлет за обработка, како и негово терминирање што се случува во моментот кога тоа ќе биде обработено, пред да се испрати одговорот на веб серверот. Во ваквите моменти контејнерот емитува настан кој е инстанца од типот `ServletRequestEvent`. Оваа класа ги содржи методите

```
public ServletContext getServletContext()  
public ServletRequest getServletRequest()
```

за пристап до контекстот и до самото барање потребни при обработка на настани.

Слушачот на настани поврзани со промената на состојбата на барањето се дефинира во класа која го имплементира интерфејсот `ServletRequestListener` со методите:

```
public void requestInitialized(ServletRequestEvent e)  
public void requestDestroyed(ServletRequestEvent e)
```

Со оглед на тоа што барањето може да содржи атрибути кои му се додадаваат во рамките на апликацијата, постои настан кои се емитува при промена на состојбата на атрибутите. Претставник на настанот за промена на атрибутите е класата `ServletRequestAttributeEvent` со методи за пристап до името и вредноста на променетиот атрибут

```
public String getName()  
public Object getValue()
```

Интерфејсот кој треба ада го имплементира слушачот на настани за промена на атрибути на барање е `ServletRequestAttributeListener` кој ги дефинира методите:

```
public void attributeAdded(ServletRequestAttributeEvent e)  
public void attributeRemoved(ServletRequestAttributeEvent e)  
public void attributeReplaced(ServletRequestAttributeEvent e)
```

и кои се поврзани со Првиот методот се повикува веднаш откако ќе се иницијализира контекстот. Во овој момент сите контекстни параметри се веќе прочитани од `web.config` и нивните референци се сместени во контекстот. Настанот се користи за да се креираат атрибути и да се постават нивните почетни вредности или пак за да се иницијализираат компоненти кои ќе ги користат сите сервлети како што е на пример конекција до база на податоци. Вториот метод се повикува пред контекстот да се избрише и се користи за да се зачуваат атрибутите во постојано складиште (на пример, во база) пред да се избришат или за да се ослободат ресурси на компонети кои ги користат сите сервлети како што е на пример затоврање на конекција до база на податоци. И двата метода како аргумент го пренесуваат настанот дефиниран со класата `ServletContextEvent` преку кој се овозможува пристап до контекстот, а со тоа и до неговите параметри и аргументи.

Имињата на класите за двата дефинирани слушачи за опслужување на настани поврзани со промена на барање или атрибути на барање треба да се конфигурира во `web.config` како нови `<listener>` јазли.

Б.8 Животен век на сесија

Предефинирираниот животен век на сесисијата е 30 минути, но тоа може да се промени во конфигурациската датотека `web.xml` со поставување на нова вредност на `<session-timeout>` во јазолот `<session-config>`. На пример, за да се постави траењето на сесија од еден час, потребно е да се вметне следниот код:

```
<session-config>
    <session-timeout>60</session-timeout>
</session-config>
```

Ова е единствениот начин да се постави глобална вредност за животниот век на сесијата на ниво на апликација. Постои и начин да се постави траењето на индивидуална сесија во код, а тоа е со повикување на методот од интерфејсот `HttpSession`

```
void setMaxInactiveInterval(int intervalInSeconds)
```

Ако се постави вредност 0, сесијата веднаш истекува, а ако се постави на вредност -1, тогаш сесијата никогаш не истекува, односно, серверот постојано го чува сесискиот објект.

За разлика од поставувањето на глобалниот животен век во `web.xml` кое е во минути, кај индивидуалното поставување преку код интервалот е во секунди.

Времето на истекување за секоја сесија се смета од последната активност на корисникот. Откако сесијата ќе истече, се брише сесискиот објект на серверот и при следното барање од истиот клиент ќе се креира нов сесиски објект при повик на `getSession()`.

Дополнително, корисникот може да го избрише сесискиот објект и да го острани колачето од оговорот пред да истече неговиот животен век преку методот:

```
public void invalidate()
```

Б.9 Пристап до параметри за сесија

Интерфејсот `HttpSession` нуди и други корисни методи за читање на информацији поврзани за сесијата како што се сесиски идентификатор, време на креирање, последен пат кога корисникот испратил барање поврзано со таа сесија, максималното време во секунди кое може корисникот да биде неактивен, а сепак да му е валидна сесијата и дали сесијата е нова (што туку креирана) или постоечка):

```
public String getId()
public long getCreationTime()
public long getLastAccessedTime()
void setMaxInactiveInterval()
public boolean isNew()
```

Б.10 Настани и слушачи на настани за сесии

Аналогно на досегашните настани и слушачи кај контекстот и барањето, постојат настани кои се емитуваат при промена на состојбата на сесијата, односно нејзино креирање или терминирање или при промена на сесиските атрибути. Класата која одговара на настанот на промена на сесијата е `HttpSessionEvent` и го нуди методот

```
public HttpSession getSession()
```

за пристап до променетата сесија, додека пак класата која одговара на настанот на промена на атрибутите е `HttpSessionBindingEvent` со методите за пристап до името на променетиот атрибут, неговата вредност и сесијата на која се однесува:

```
public String getName()
public Object getValue()
public HttpSession getSession()
```

Соодветно, за обработка на настанот кој одговара на HttpSessionEvent се користи интерфејсот HttpSessionListener со методите

```
public void sessionCreated(HttpSessionEvent e)  
public void sessionDestroyed(HttpSessionEvent e)
```

а за настанот кој одговара на HttpSessionBindingEvent се користи интерфејсот HttpSessionAttributeListener со методите

```
public void attributeAdded(HttpSessionBindingEvent e)  
public void attributeRemoved(HttpSessionBindingEvent e)  
public void attributeReplaced(HttpSessionBindingEvent e)
```

Имињата на класите за двета дефинирани слушачи за опслужување на настани поврзани со промена на барање или атрибути на барање треба да се конфигурира во web.config како нови <listener> јазли.

Додаток В

Lombok

Пакетот *Lombok*¹ овозможува анотирање на класи за автоматско генерирање на пропратен код за класи кој ја зголемува читливоста на класите, но истовремено и продуктивноста на програмерите. Една од главните анатации на пакетот е анатацијата `@Data` која ги обединува на следните поедични анатации за автоматско генерирање на методи:

- `@Getter` за читање на својства
- `@Setter` за поставување на својства
- `@RequiredArgsConstructor` конструктор со аргументи за оние својства кои се од типот `final` или се означени со анатацијата `@NonNull` како својства кои не можат да имаат вредност `null`, но не се претходно иницијализирани при декларацијата
- `@ToString` за конверзија на инстанците од класата во текстуална низа која ги вклучува имињата на својствата и нивните вредности
- `@EqualsAndHashCode` за споредба на инстанци и генерирање нивно резиме (hash)

Било која анатација може да биде препокриена доколку програмерот во кодот рачно генерира соодветен метод. На пример, ако корисникот самиот креира метод за читање на својството `name` со поптис `String getName()`, тогаш Lombok нема да креира метод за читање и ќе се користи кориснички-дефинираниот метод `getName()`. За овие анатации да се валидни, потребно е да се вклучи следниот пакет: `import lombok.Data;`

Анатацијата `@Data` не вклучува автоматско генерирање на конструктор без аргументи, што е еден од основните услови за дефиниција на ентитети, па затоа е потребно или да се додаде Lombok анатацијата `@NoArgsConstructor` или мануелно да се креира.

¹<https://projectlombok.org/>