



Универзитет „Св. Климент Охридски” – Битола

Факултет за информатички и комуникациски

технологии



-СЕМИНАРСКА РАБОТА-

Предмет: Тестирање на софтвер, обезбедување квалитет и одржување

Автоматско тестирање со помош на Postman

Ментор:

Проф. д-р Илија Јолевски

Кандидат:

Вероника Спасевска ИНКИ56

Содржина

Вовед	2
Тестирање на софтвер	3
Функционално Тестирање	4
Нефункционално Тестирање	5
Рачно Тестирање	6
Автоматско Тестирање	6
Придобивки од автоматското тестирање	8
Лоши страни на автоматското тестирање	8
Како се носи одлука кој тест да се автоматизира?	9
Тестови кои обично се автоматизирани	9
Интеграциско Тестирање	12
Историја на софтверското тестирање	13
Зошто е важно тестирањето на софтверот?	14
Најдобри практики за тестирање на софтвер	15
Continuous testing	16
Configuration management	16
Service virtualization	17
Defect or bug tracking	17
Metrics and reporting	17
Postman	18
Историја на Postman	18
Карактеристики на Postman	18
Практичен пример	19
Заклучок	53
Користена литература	54

Вовед

Тестирањето на софтверот е клучен аспект во развојот на апликации и системи, кој ја осигурува нивната функционалност, безбедност и квалитет. Во оваа семинарска работа, ќе ги разгледаме различните типови на тестирање, вклучувајќи функционално и нефункционално тестирање, како и значењето на рачното и автоматското тестирање. Автоматизацијата на тестовите нуди бројни придобивки, но и предизвици.

Во оваа семинарска работа ќе извршиме автоматско тестирање на апликација креирана со помош на Spring Boot. Тестирањето ќе го извршиме во Postman.

Целта е да се обезбеди целосен преглед на важноста на тестирањето во развојот на софтвер, како и да се истакне улогата во обезбедувањето на квалитетни производи кои ги исполнуваат потребите на корисниците.

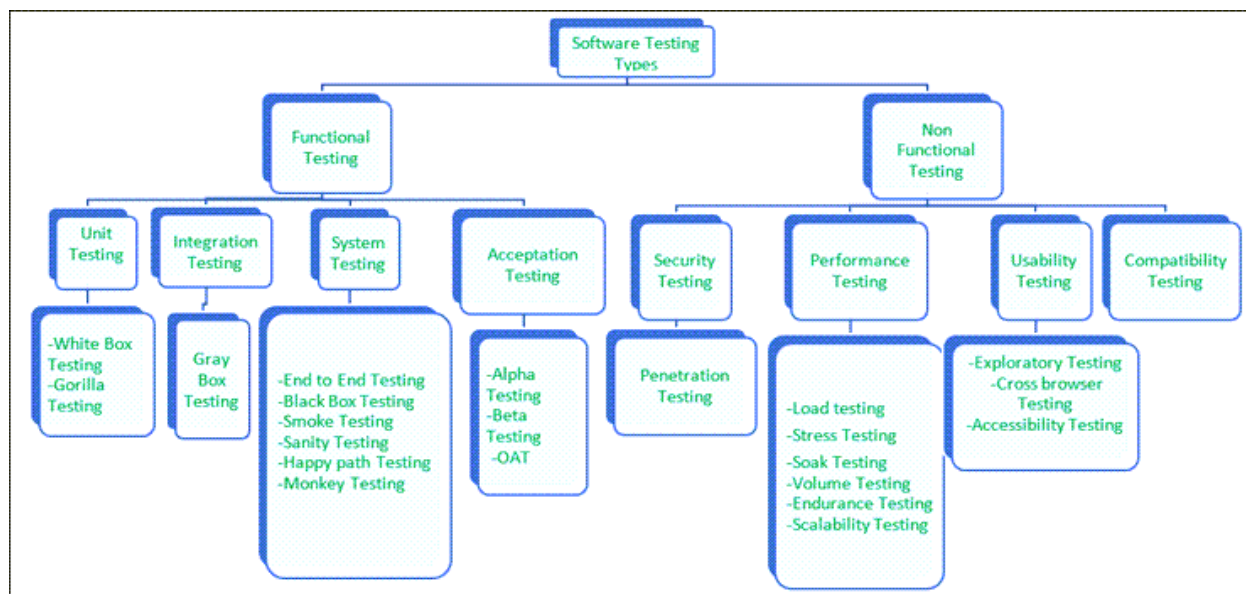
Тестирање на софтвер

Тестирањето на софтвер е процес на оценување и потврдување дали софтверскиот производ или апликација го прави она што треба да го прави. Придобивките од доброто тестирање вклучуваат спречување на грешки и подобрување на перформансите.

Тестирањето на софтверот денес е најефективно кога е континуирано, што покажува дека тестирањето започнува за време на дизајнот, продолжува додека софтверот се изградува, па дури и се случува кога се распоредува во производство. Континуираното тестирање значи дека организациите не мора да чекаат да се распоредат сите делови пред да започне тестирањето. Shift-left, што го придвижува тестирањето поблиску до дизајнот и shift-right, каде што крајните корисници вршат валидација, се исто така филозофии на тестирање кои неодамна добиле привлечност во софтверската заедница. Кога ќе се разберат стратегијата за тестирање и плановите за управување, автоматизацијата на сите аспекти на тестирањето станува од суштинско значење за поддршка на брзината на испораката што е потребна.

Видови на софтверско тестирање

Во продолжение е класификацијата од високо ниво за типови за тестирање на софтвер.



Сл.1 Видови на тестирање на софтвер (Извор: <https://www.softwaretestinghelp.com/types-of-software-testing/>)

Како што можеме да забележиме највисоко ниво се **функционални** и **нефункционални** типови на тестирање, од кои се разгрануваат и останатите поделби. Во продолжение ќе бидат објаснети највисоките гранки во хиерархијата (**функционално** и **нефункционално** тестирање) и интеграциско тестирање. Според начин на извршување на тестовите тие се делат на **рачни** и **автоматизирани** тестови, кои исто така ќе бидат опфатени во продолжение.

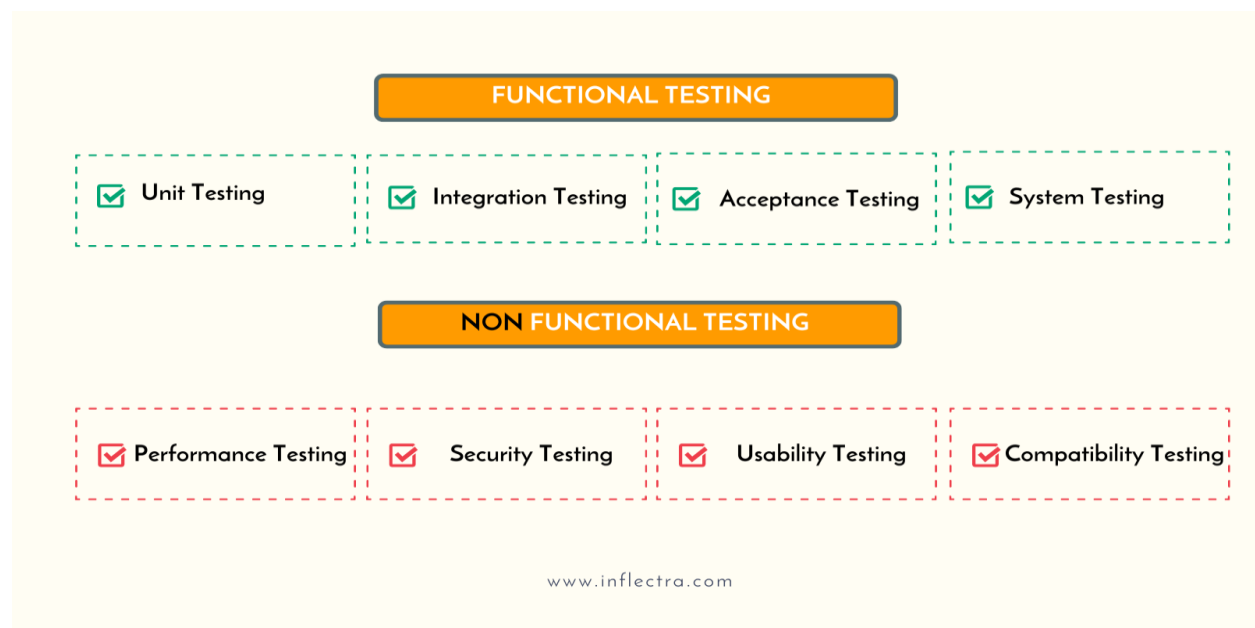
Функционално Тестирање

Функционалното тестирање е вид на тестирање кое потврдува дека секоја функција на софтверската апликација работи во согласност со спецификацијата за барањата. Секоја функционалност на системот се тестира со обезбедување соодветен влез, верификација на излезот и споредување на вистинските

резултати со очекуваните резултати. Функционалното тестирање се нарекува и тестирање на црна кутија, бидејќи се фокусира на спецификацијата на апликацијата наместо на вистинскиот код. Тестерот треба да ја тестира само програмата наместо системот.

Нефункционално Тестирање

Нефункционалното тестирање е вид на тестирање за проверка на нефункционалните аспекти (перформанси, употребливост, доверливост итн.) на софтверска апликација. Тој е експлицитно дизајниран да ја тестира подготвеноста на системот според нефункционалните параметри кои никогаш не се решаваат со функционално тестирање.



Сл.2 Функционално и нефункционално тестирање (Извор:

<https://www.inflectra.com/Ideas/Topic/Functional-vs-Non-Functional-Testing.aspx>)

Нефункционалното тестирање дава детално знаење за однесувањето на производот и користените технологии. Тоа помага во намалување на ризикот од производство и придружните трошоци на софтверот.

Рачно Тестирање

Рачно тестирање е тестирање на софтверот каде тестовите се извршуваат рачно од аналитичар за **QA**. Се изведува за да се откријат грешки во софтверот во развој.








Во **рачно тестирање**, тестерот ги проверува сите суштински карактеристики на дадената апликација или софтвер. Во овој процес, софтверските тестери ги извршуваат **случаите за тестирање** и ги генерираат извештаите за тестирање без помош на алатки за тестирање на софтвер за автоматизација.

Автоматско Тестирање

Во **автоматско тестирање** на софтвер, тестерите пишуваат кодови/тест **скрипти** за да го **автоматизираат** извршувањето на тестот. **Тестерите** користат соодветни алатки за автоматизација за да ги развијат тест скриптите и да го потврдат **софтверот**. Целта е да се заврши процесот на извршување на тестот за помалку време.

Автоматското тестирање целосно се потпира на претходно напишаниот тест кој се извршува автоматски за да се спореди вистинскиот резултат со

очекуваните резултати. Ова му помага на тестерот да утврди дали апликацијата работи како што се очекува или не.

Automation testing	
 Fast	 Reusable
 Reliable	 Improves accuracy
 Saves time and money	 Reduces human-generated error
 Supports the execution of repeated test cases	

Сл.3 Автоматско тестирање (Извор: <https://www.globalapptesting.com/blog/what-is-automation-testing>)

Додека сите тестирања, вклучително и регресивното тестирање, може да се направат рачно, често има поголеми придобивки од автоматско извршување на голем дел од тоа. Автоматското тестирање овозможува флексибилност бидејќи:

- Тестовите може да се извршат во секое време
- Тое е побрзо
- Тое е исплатливо
- Им овозможува на квалитетните инженери да се справат со поголем волумен на тест од рачно тестирање

За многу професионалци за обезбедување квалитет, транзицијата кон кариерата на инженер за автоматизација станува изводлива. Инженерите особено веруваат дека тестовите “треба” да се автоматизираат, дури и кога деловната реалност е многу понеуредна.

Програмерите кои пишуваат автоматски тестови обично пишуваат на C#, JavaScript и Ruby како програмски јазици. Многу алатки можат да помогнат во пишувањето автоматизирани тестови и да помогнат во управувањето со нив.

Придобивки од автоматското тестирање

Иако брзината останува примарна придобивка од автоматското тестирање, се појавуваат и други предности кои ја подобруваат ефикасноста на процесот на развој на софтвер. Некои од придобивките од автоматското тестирање се:

- Инстант враќање на тестот и заштедени работни часови
- Брзи повратни информации
- Оптимална распределба на ресурсите
- Зголемена точност
- Проширена покриеност на тестот
- Рано откривање на грешки
- Скалабилно тестирање

Лоши страни на автоматското тестирање

Негативни страни на автоматското тестирање вклучуваат:

- Висока почетна цена
- Неможност да се замени човечката интуиција
- Maintenance overhead
- Ограничено тестирање за корисничко искуство
- Предизвици со сложени сценарија
- Креирањето на тест-скрипти одзема многу време

- Неможност да се прилагодат на честите промени на UI
- Ризик од лажни позитиви и негативи
- Зависност од вештини
- Преценета автоматизација

Како се носи одлука кој тест да се автоматизира?

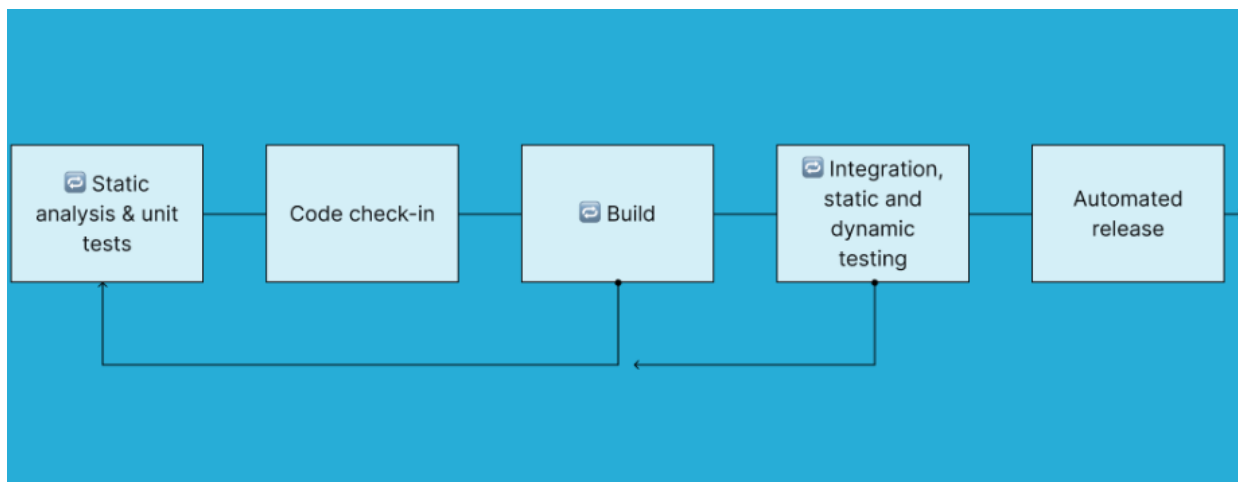
Од суштинско значење е да се даде приоритет на автоматизацијата на тестовите врз основа на нивната пожелност, бидејќи постои можност да не може да се автоматизира секој тест. Шеми според кои најдобро е да се направи изборот:

- Регресивни тестови
- Тест-случаи кои се временски интензивни или треба често да се повторуваат
- Тестови кои можат да доведат до неуспех поради човечка грешка
- Повторливи и монотони тестови
- Обемни тестови кои бараат повеќе збирки на податоци
- Тестови кои не можат да се извршат рачно
- Тестови со висок ризик

Тестови кои обично се автоматизирани

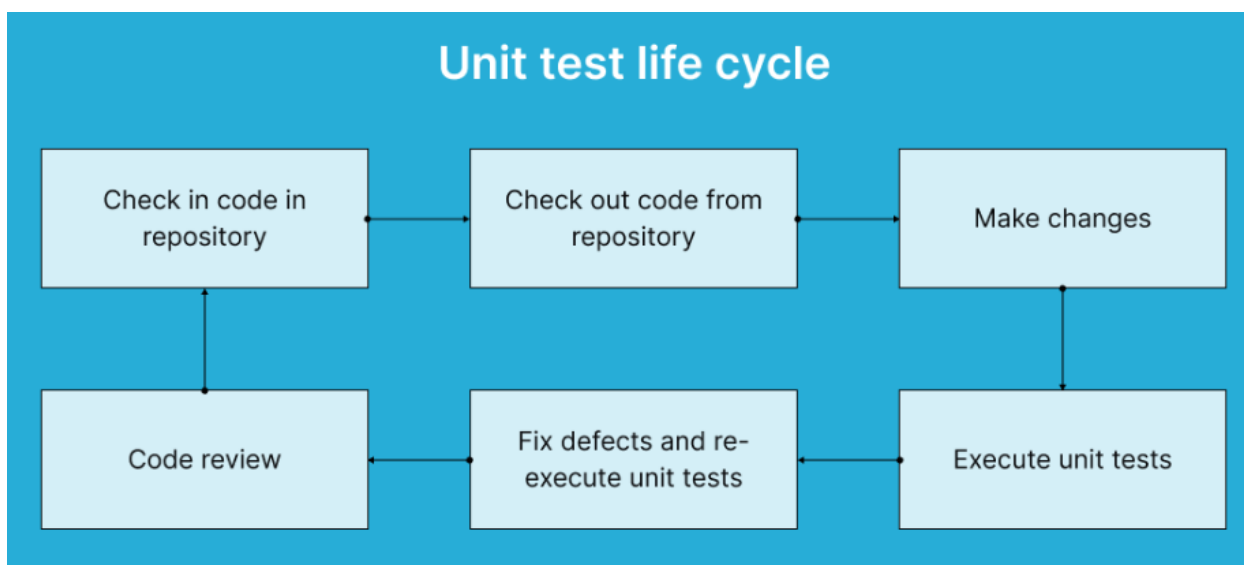
Иако повеќето тестови можат да се автоматизираат, ова се најчестите категории на автоматизирани тестови:

- Анализа на кодови



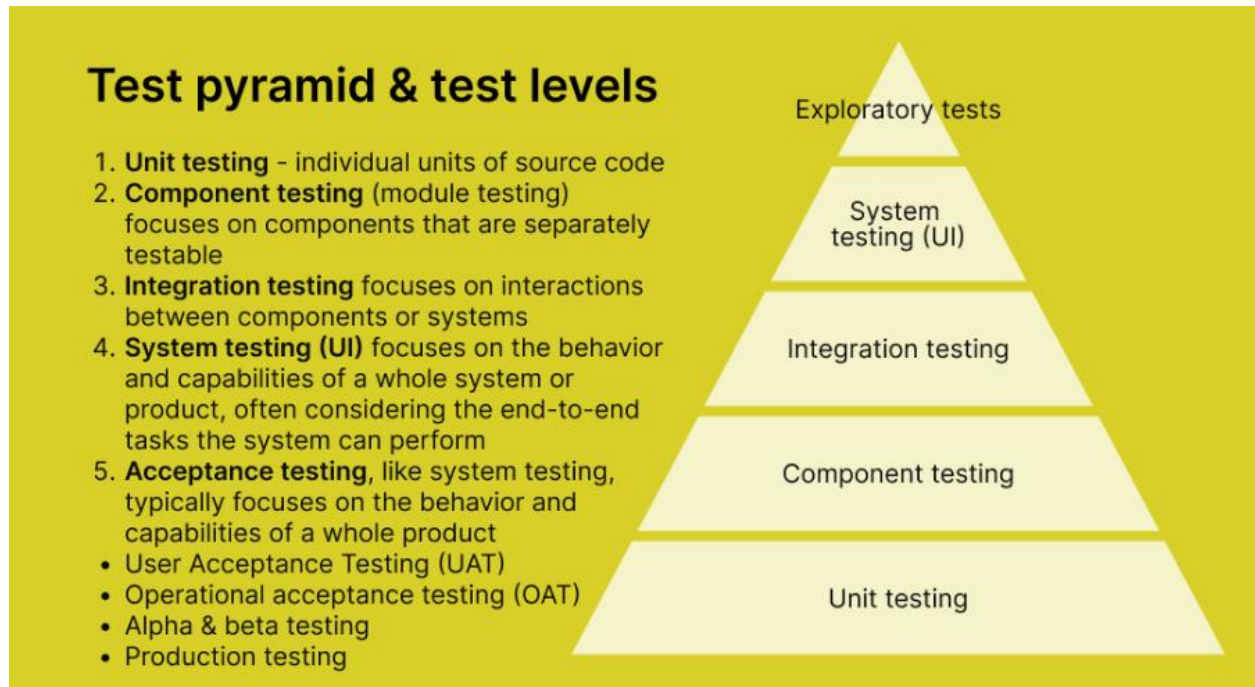
Сл.4 Анализа на кодови (Извор: <https://www.globalapptesting.com/blog/what-is-automation-testing>)

- Unit тестови



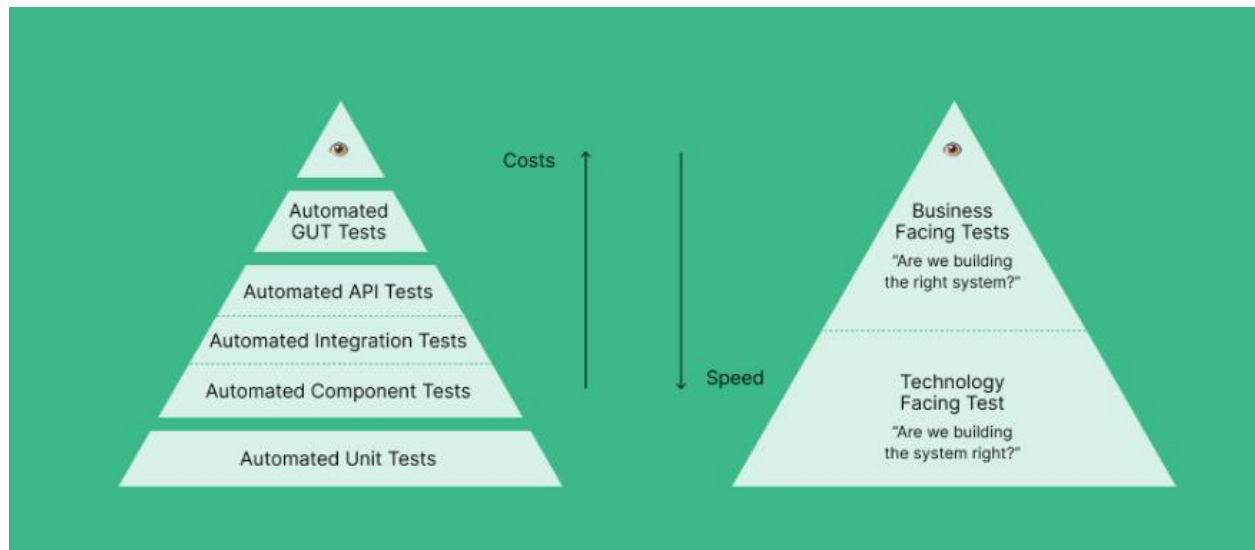
Сл.5 Животен циклус на Unit Test (Извор: <https://www.globalapptesting.com/blog/what-is-automation-testing>)

- Интеграциски тестови



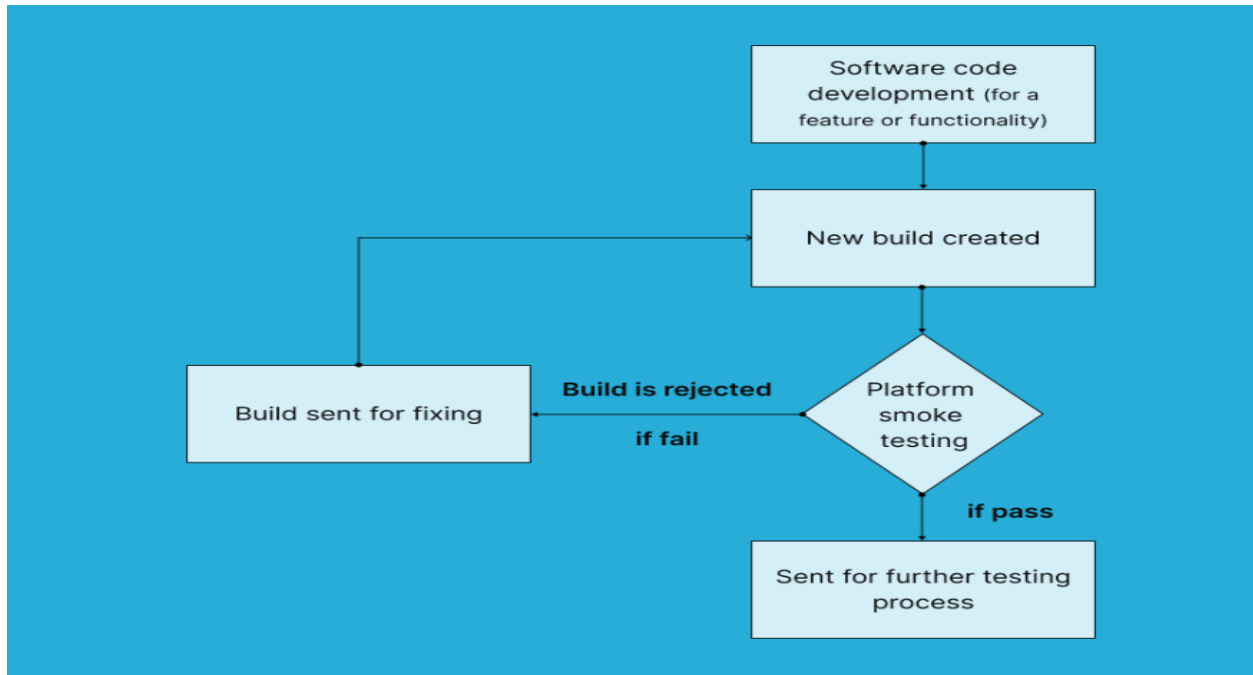
Сл.6 Тест пирамида и нивоа на тестирање (Извор: <https://www.globalapptesting.com/blog/what-is-automation-testing>)

- Automated acceptance tests



Сл.7 Automated acceptance test (Извор: <https://www.globalapptesting.com/blog/what-is-automation-testing>)

- smoke tests



Сл.8 Smoke tests (Извор: <https://www.globalapptesting.com/blog/what-is-automation-testing>)

Интеграциско Тестирање

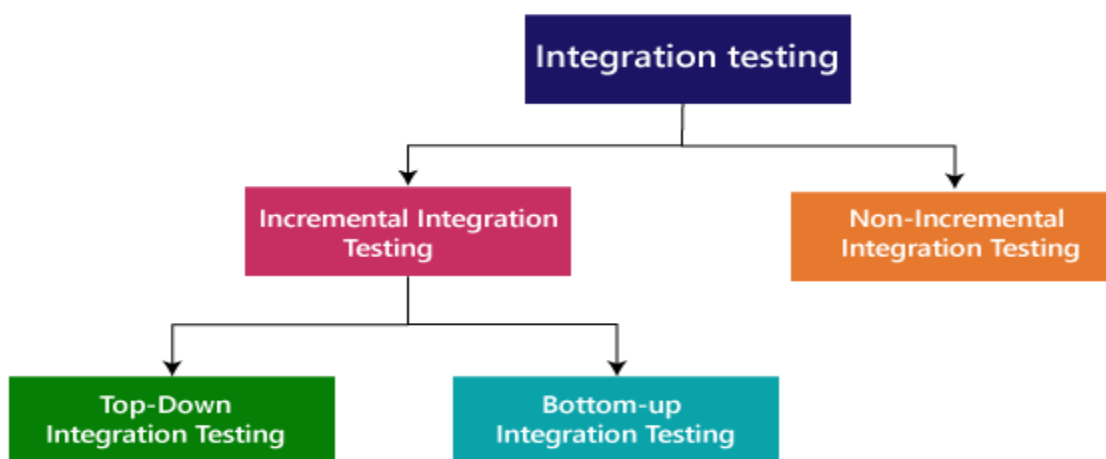
Тестирањето за интеграција е дефинирано како тип на тестирање каде софтверските модули се **интегрираат** логично и се тестираат како **група**. Типичен софтверски проект се состои од повеќе софтверски модули, кодирани од различни програмери. Целта на ова ниво на тестирање е да ги открие дефектите во интеракцијата помеѓу овие софтверски модули кога тие се интегрирани.

Видови на интеграциско тестирање:

Софтверското инженерство дефинира различни стратегии за извршување на тестирање за интеграција:

- **Биг Бенг Пристап**

- **Инкрементален пристап:** кој понатаму е поделен на
- **Пристап од горе надолу**
- **Пристап одоздола нагоре**
- **Сендвич Пристап** - комбинација од горе надолу и долу нагоре



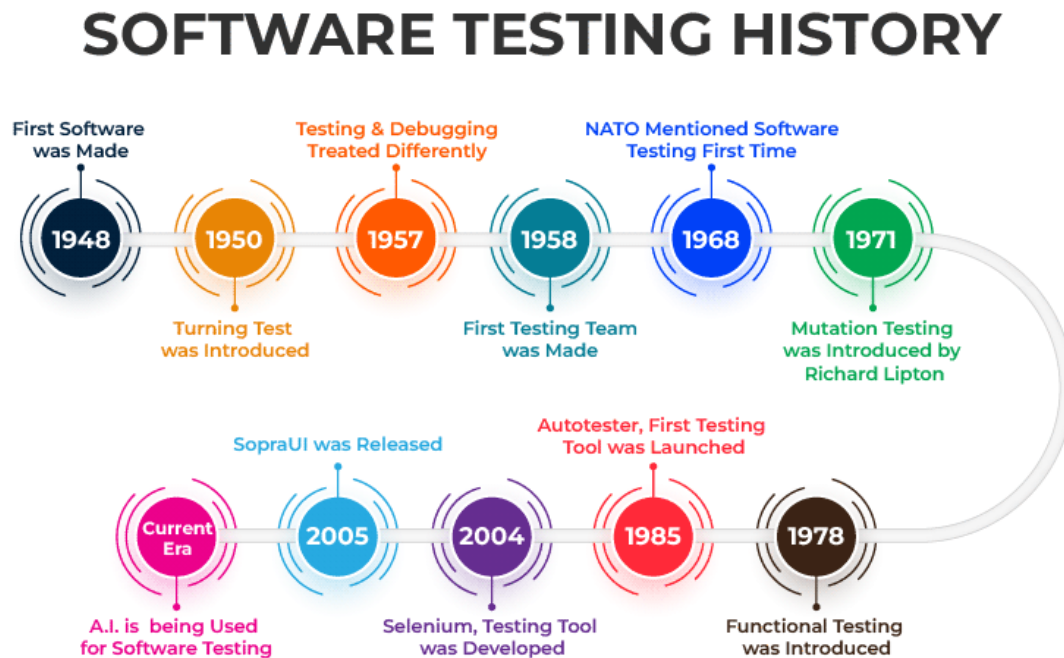
Сл.4 Интеграциско тестирање (Извор: <https://www.javatpoint.com/integration-testing>)

Историја на софтверското тестирање

Софтверското тестирање пристигна заедно со развојот на софтверот, кој ги имаше своите почетоци веднаш по Втората Светска Војна. Компјутерскиот научник Том Килбурн е заслужен за пишувањето на првото парче софтвер, кое дебитирање на 21 јуни 1948 година на Универзитетот во Манчестер. Тоа парче софтвер вршеше математички пресметки користејќи инструкции за машински код.

Дебагирањето беше главниот метод на тестирање во тоа време и остака така во следните две децении. До 1980-тите, развојните тимови гледаа подалеку од изолирање и поправање на софтверски грешки за тестирање на апликации во

реални поставки. Тој ја постави основата за поширок поглед на тестирањето, кое опфаќаше процес на обезбедување квалитет кој беше дел од животниот циклус на развој на софтвер.



Сл.7 Историја на тестирање на софтвер (Извор: <https://www.geeksforgeeks.org/history-of-software-testing/>)

Зошто е важно тестирањето на софтверот?

Малкумина можат да се расправаат против потребата за контрола на квалитетот при развој на софтвер. Доцна испорака или дефекти на софтверот може да ја нарушат репутацијата на брендот, што доведува до фрустрирани и изгубени клиенти. Во екстремни случаи, грешка или дефект може да ги деградира меѓусебно поврзаните системи или да предизвика сериозни дефекти.

Иако самото тестирање чини пари, компаниите можат да заштедат милиони годишно во развојот и поддршката доколку имаат добра техника за тестирање и воспоставени QA процеси. Раното тестирање на софтверот открива проблеми пред производот да излезе на пазарот. Колку побрзо развојните тимови добиваат повратни информации од тестот, толку побрзо ќе можат да ги решат проблемите како што се:

- Архитектонски недостатоци
- Лоши одлуки за дизајн
- Невалидна или неточна функционалност
- Безбедносни пропусти
- Проблеми со приспособливост

Кога развојот остава доволно простор за тестирање, тој ја подобрува доверливоста на софтверот и висококвалитетните апликации се испорачуваат со малку грешки. Системот што ги исполнува или дури ги надминува очекувањата на клиентите води до потенцијално поголема продажба и поголем удел на пазарот.

Најдобри практики за тестирање на софтвер

Тестирањето на софтверот следи заеднички процес. Задачите или чекорите вклучуваат дефинирање на околината за тестирање, развивање тест случаи, пишување скрипти, анализа на резултатите од тестот и поднесување извештаи за дефекти.

Тестирањето може да одземе многу време. Рачно тестирање или ад хок тестирање може да биде доволно за мали изданија. Меѓутоа, за поголеми системи,

алатките често се користат за автоматизирање на задачите. Автоматското тестирање им помага на тимовите да имплементираат различни сценарија, да тестираат диференцијатори(како што се преместување на компоненти во cloud околина) и брзо да добијат повратни информации за тоа што функционира, а што не.

Добриот пристап за тестирање ги опфаќа интерфејсот за програмирање на апликации, корисничкиот интерфејс и нивоата на системот. Колку повеќе тестови се автоматизирани и се извршуваат рано, толку подобро. Некои тимови градат build in-house алатки за автоматско тестирање. Сепак, решенијата на продавачите нудат карактеристики што можат да ги насочат клучните задачи за управување со тестови, како што се:

Continuous testing

Проектните тимови ја тестираат секоја верзија кога ќе стане достапна. Овој тип на тестирање на софтвер се потпира на автоматизација на тестот што е интегрирана со процесот на распоредување. Овозможува валидација на софтверот во реални тест околина порано во процесот, што го подобрува дизајнот и ги намалува ризиците.

Configuration management

Организациите централно ги одржуваат тест-средствата и следат што софтверот создава за тестирање. Тимовите добиваат пристап до средства како код, барања, документи за дизајн, модели, тест скрипти и резултати од тестот. Добрите системи вклучуваат автентикација на корисникот и ревизорски патеки за да им

помогнат на тимовите да ги исполнат барањата за усогласеност со минимален административен напор.

Service virtualization

Околините за тестирање можеби не се достапни, особено во почетокот на развојот на кодот. Виртуелизацијата на услугите ги симулира услугите и системите кои недостасуваат или се уште не се завршени, овозможувајќи им на тимовите да ги намалат зависностите и да тестираат порано. Тие можат повторно да користат, распоредуваат и менуваат конфигурација за да тестираат различни сценарија без да мора да ја менуваат оригиналната околина.

Defect or bug tracking

Следењето на дефектите е важно и за тимовите за тестирање и за развој за мерење и подобрување на квалитетот. Автоматските алатки им овозможуваат на тимовите да ги следат дефектите, да го мерат нивниот опсег и влијание и да откријат поврзани проблеми.

Metrics and reporting

Известувањето и аналитиката им овозможуваат на членовите на тимот да споделуваат статус, цели и резултати од тестовите. Напредните алатки ги интегрираат метриците на проектот и ги прикажуваат резултатите во контролната табла. Тимовите брзо го гледаат целокупното здравје на проектот и можат да ги следат односите помеѓу тестот, развојот и другите елементи на проектот.

Postman

Postman е API платформа за градење и користење на APIs. Го поедноставува секој чекор од животниот циклус на API и ја рационализира соработката за да можат да се креираат подобри APIs – побрзо. Постојат над 30 милиони регистрирани корисници и околу 500 000 организации кои го користат Postman. Postman исто така ја одржува Postman API мрежата, directory со повеќе од 100 000 јавни APIs, која е наведена како најголема таква мрежа во светот. Компанијата е со седиште во Сан Франциско и има дополнителни канцеларии на повеќе локации во светот.

Историја на Postman

Postman започнал во 2012 година како спореден проект на софтверскиот инженер Abhinav Asthana, кој сакал да го поедностави тестирањето на API додека работел во Yahoo. Тој ја нарекол својата апликација Postman – игра на API request POST и ја понудил бесплатно во Chrome Web Store. Како што употребата на апликацијата растела, Абхинав ги регрутирал поранешните колеги за да помогнат во создавањето на Postman, Inc. Тројцата коосновачи ја водат компанијата денес, со Абхинав како CEO на компанијата. Во 2023 година, Postman објавил дека ја купил Akita, решение за набљудување на API. Во 2024 година Postman го купил Orbit, решение за градење и управување со заедниците на програмери.

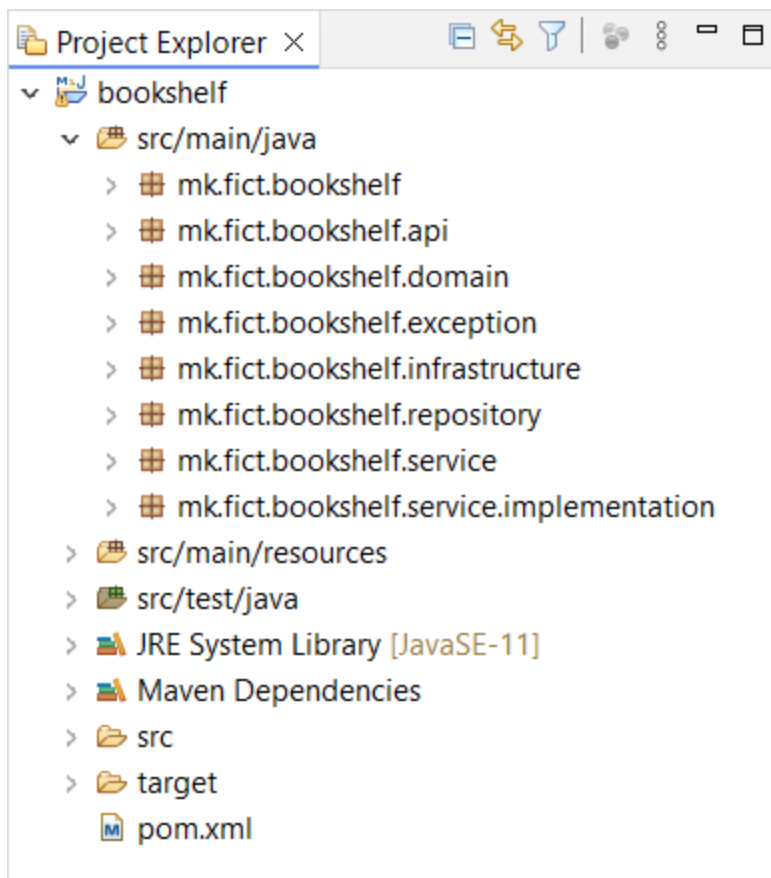
Карактеристики на Postman

Карактеристиките на Postman вклучуваат:

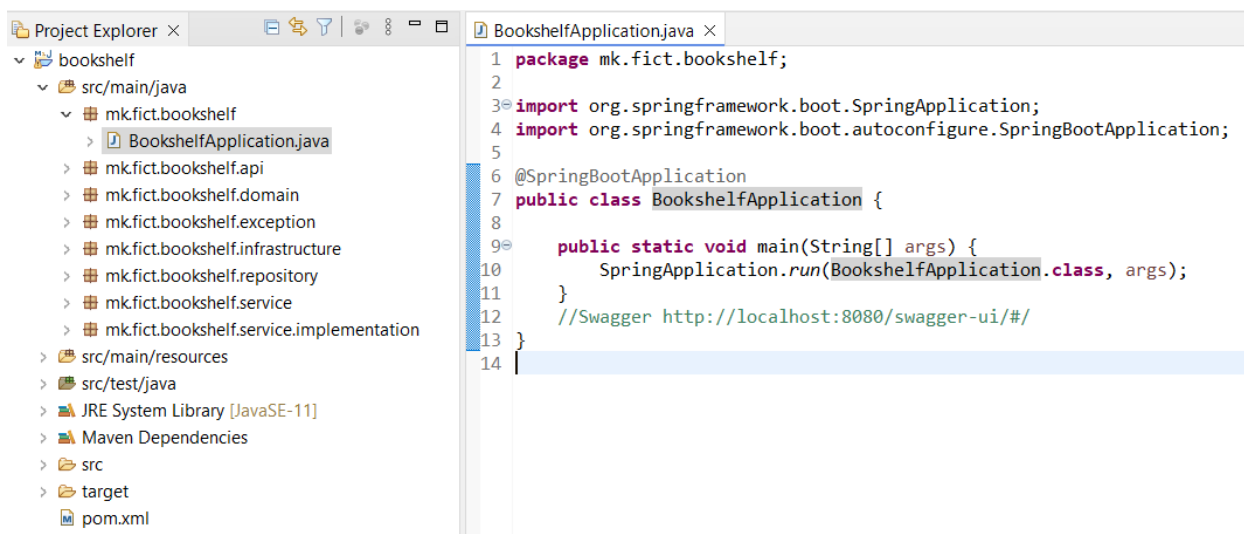
- **Workspaces** (работни простори): личните, тимските, партнерските и јавните работни простори овозможуваат соработка со API внатрешно и надворешно
- **API repository** (складиште на API): им овозможува на корисниците да складираат, каталогизираат и соработуваат околу артефактите на API во централна платформа во јавни, приватни или партнерски мрежи
- **API Builder**: Помага во спроведувањето на работниот тек на дизајнот на API преку спецификации вклучувајќи OpenAPI, GraphQL и RAML. Интегрира различни контроли на изворот, CI/CD, портали и APM решенија
- **Tools** (алатки): API клиент, API дизајн, API документација, API тестирање, лажни сервери и API откривање
- **Intelligence**: Безбедносни предупредувања, пребарување на складиштето на API, работни простори, известување, управување со API.

Практичен пример

За да се извршат автоматските тестови, креирана е Spring Boot апликација за bookshelf. Класите се сместени во повеќе пакети, како што е прикажано на сликата 8. Во овој проект опфатени се повеќе ентитети: Author, Book, DigitalSource, Genre, Publisher, но заради намалување на обемот на семинарската работа, ќе бидат прикажани само Author и Book.



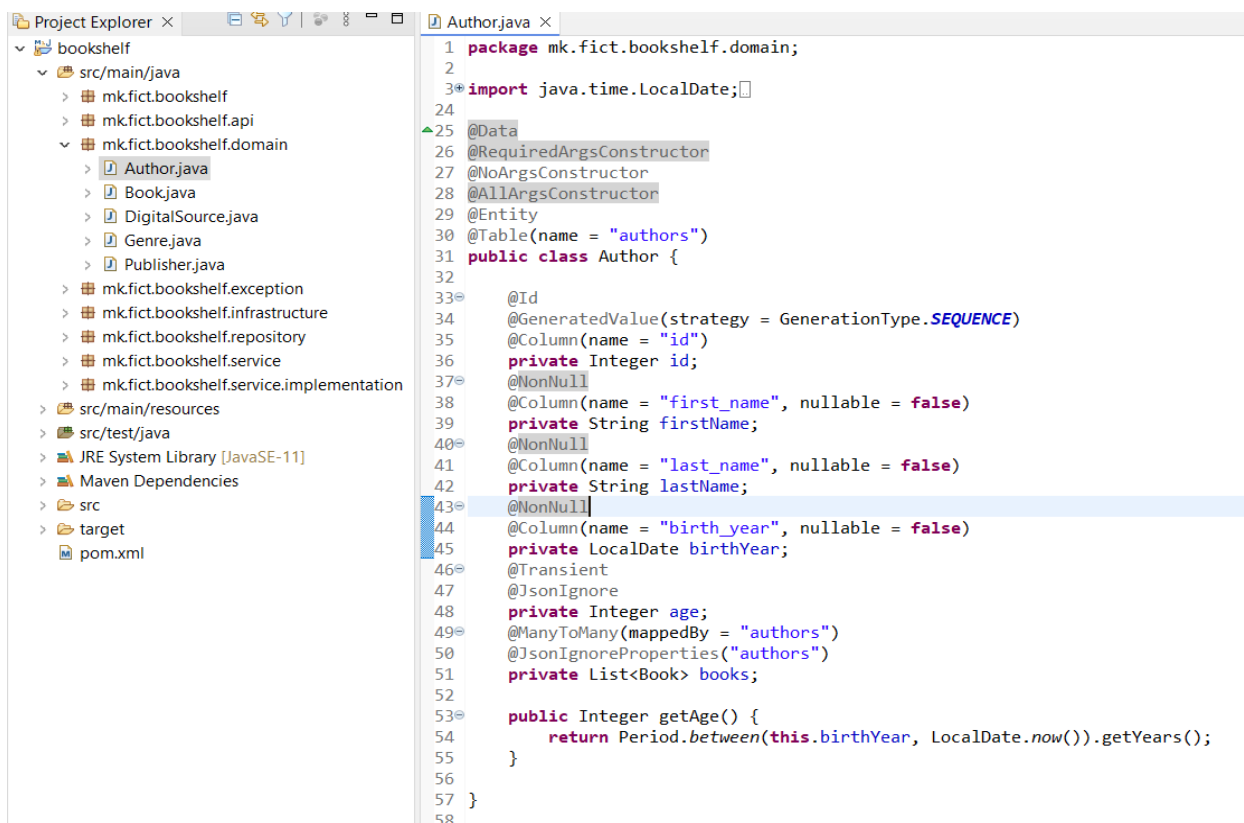
Сл.8 Структурата на bookshelf проектот



Сл.9 Класата Bookshelf Application во пакетот mk.fict.bookshelf

Во пакетот `mk.fict.bookshelf` креираме класа `BookshelfApplication` која претставува едноставна Spring Boot апликација. Прво го внесуваме името на пакетот, а после тоа потребните imports од Spring Boot framework. После тоа внесуваме анотација `@SpringBootApplication`. Понатаму, се внесува главниот (main) метод кој е главна точка на апликацијата. `SpringApplication.run (...)` се грижи за поставување на контекстот на апликацијата, вчитување на конфигурацијата и стартување на вградениот сервер.

Во пакетот `mk.fict.bookshelf.domain` е прикажано креирањето на табелите и колоните во базата како и врските – many to many, one to many.. На сликата подолу е претставен кодот за ентитетот `Author`.



```
1 package mk.fict.bookshelf.domain;
2
3 import java.time.LocalDate;
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25 @Data
26 @RequiredArgsConstructor
27 @NoArgsConstructor
28 @AllArgsConstructor
29 @Entity
30 @Table(name = "authors")
31 public class Author {
32
33     @Id
34     @GeneratedValue(strategy = GenerationType.SEQUENCE)
35     @Column(name = "id")
36     private Integer id;
37     @NotNull
38     @Column(name = "first_name", nullable = false)
39     private String firstName;
40     @NotNull
41     @Column(name = "last_name", nullable = false)
42     private String lastName;
43     @NotNull
44     @Column(name = "birth_year", nullable = false)
45     private LocalDate birthYear;
46     @Transient
47     @JsonIgnore
48     private Integer age;
49     @ManyToMany(mappedBy = "authors")
50     @JsonIgnoreProperties("authors")
51     private List<Book> books;
52
53     public Integer getAge() {
54         return Period.between(this.birthYear, LocalDate.now()).getYears();
55     }
56
57 }
58
```

Сл.10 Класата `Author` од пакетот `mk.fict.bookshelf.domain`

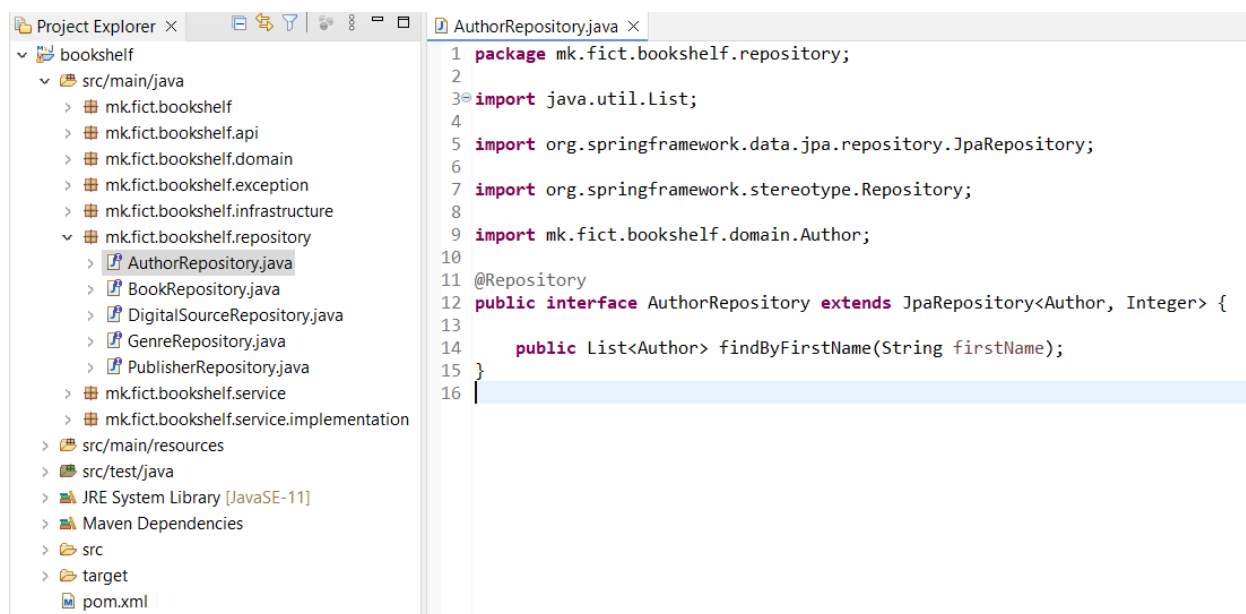
Првата линија код го претставува пакетот во кој ќе се креира оваа класа. Потоа се внесуваат потребните библиотеки, како и JPA, Jackson и Lombok анотациите. Се креира класата **Authors**, која ја претставува authors табелата во базата со соодветните полиња: **id**, **first_name**, **last_name**, **birth_year**, **age**. Со ентитетот **books** е претставена врската **many to many**. Тоа значи дека еден автор може да напише повеќе книги и една книга може да биде напишана од повеќе автори. Методот `getAge()` пресметува колку години има некој автор во зависност од внесената година на раѓање.

```
1 package mk.fict.bookshelf.domain;
2
3 import java.util.Date;
4
5 @Data
6 @RequiredArgsConstructor
7 @NoArgsConstructor
8 @AllArgsConstructor
9 @Entity
10 @Table(name = "books")
11 public class Book {
12
13     @Id
14     @GeneratedValue(strategy = GenerationType.SEQUENCE)
15     @Column(name = "id")
16     private Integer id;
17
18     @NonNull
19     @Column(name = "title", nullable = false)
20     private String title;
21
22     @NonNull
23     @Column(name = "rating", nullable = false)
24     private Integer rating;
25
26     @NonNull
27     @Column(name = "isbn", nullable = false, unique = true)
28     private String isbn;
29
30     @NonNull
31     @Column(name = "published_date", nullable = false)
32     private Date publishedDate;
33
34     @ManyToMany
35     @JoinTable(name = "books_genres", joinColumns = @JoinColumn(name = "bookid"), inverseJoinColumns = @JoinColumn(name = "genreid"))
36     private List<Genre> genres;
37
38     @ManyToMany
39     @JoinTable(name = "books_authors", joinColumns = @JoinColumn(name = "bookid"), inverseJoinColumns = @JoinColumn(name = "authorid"))
40     private List<Author> authors;
41 }
42
```

Сл.11 Класата Book од пакетот mk.fict.bookshelf.domain

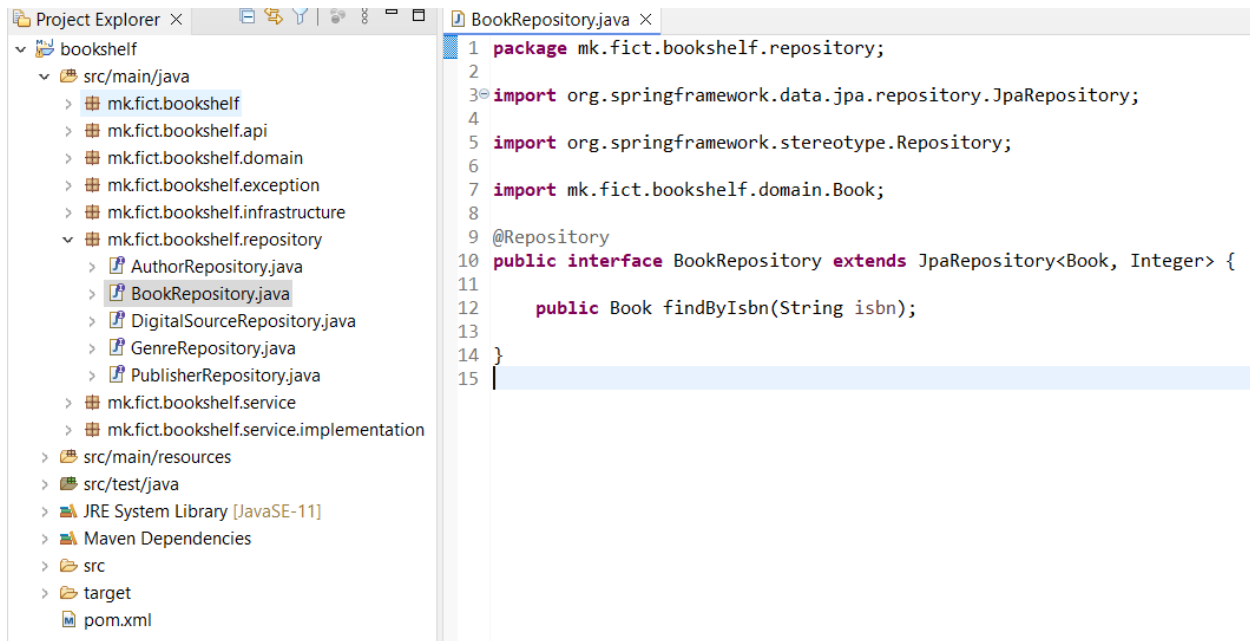
Првата линија код го покажува пакетот во кој ќе се наоѓа класата. После тоа се внесуваат соодветните imports, односно се внесуваат потребните Java библиотеки и JPA, Jackson и Lombok анотации. После тоа се декларира класата Book која ја претставува book табелата во базата. Се внесуваат соодветните полиња – колони во табелата books: id, title rating, isbn, published_date, book_genres, publisher_id и books_authors. Колоната publisher_id ја има врската many-to-one. Со ентитетот Author постои врска many-to-many, што значи дека една книга може да има повеќе автори.

Во пакетот mk.fict.bookshelf.repository имаме креирано соодветни интерфејси. На сликата подолу може да се забележи интерфејсот AuthorRepository кој го наследува JpaRepository, кој дава пристап до многу корисни методи за интеракција со базата на податоци. Прилагодениот метод findByFirstName е исто така добро дефиниран и овозможува да се пребаруваат автори според името.



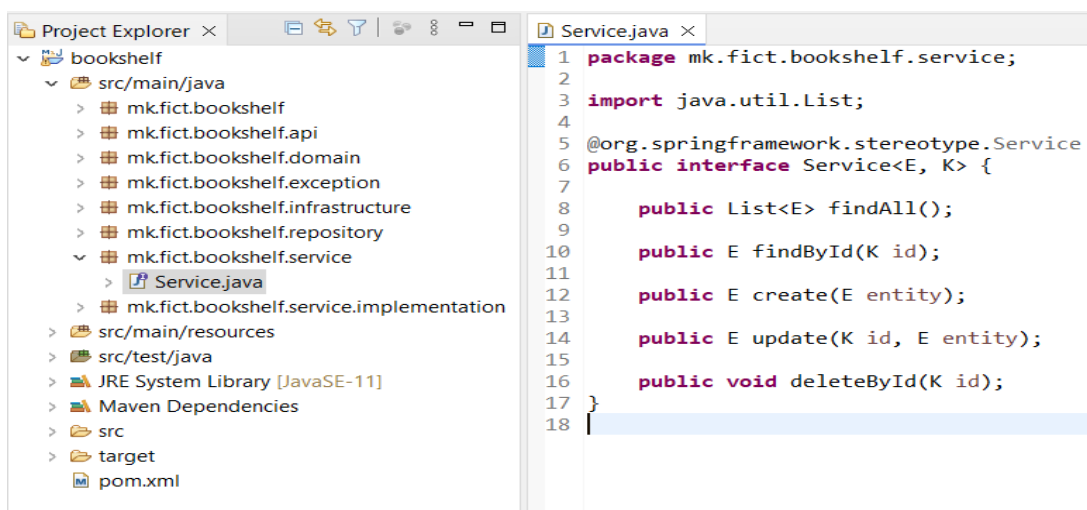
Сл.12 Интерфејсот AuthorRepository во пакетот mk.fict.bookshelf.repository

Интерфејсот BookRepository го наследува JpaRepository и вклучува приспособен метод за наоѓање книга од базата на податоци според неговиот ISBN.



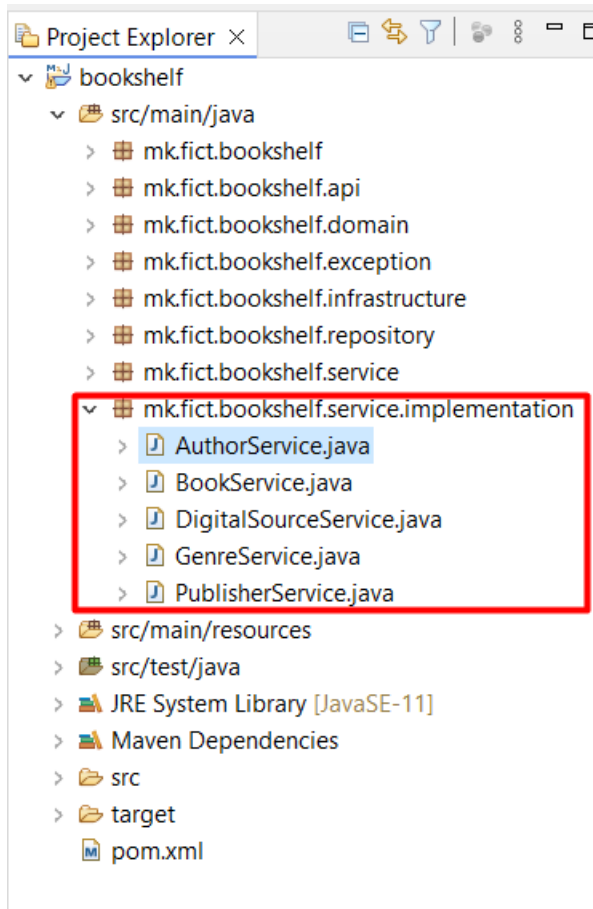
Сл.13 Интерфејсот BookRepository во пакетот mk.fict.bookshelf.repository

Во пакетот mk.fict.bookshelf.service се наоѓа генеричкиот сервисен интерфејс. Обезбедува чиста апстракција за вообичаените CRUD операции.



Сл.14 Сервис интерфејс

Во пакетот `mk.fict.bookshelf.service.implementation` се наоѓаат имплементациите на сервис интерфејсот за сите потребни ентитети.



Сл.15 сервис имплементации

Во `AuthService.java` се наоѓа класата `AuthService` која го наследува `Service` интерфејсот.

```

1 package mk.fict.bookshelf.service.implementation;
2
3 import java.util.List;
4
14
15 @org.springframework.stereotype.Service
16 @Transactional
17 public class AuthorService implements Service<Author, Integer> {
18
19     @Autowired
20     private AuthorRepository arepo;
21
22     @Override
23     public List<Author> findAll() {
24         if (arepo.findAll().isEmpty()) {
25             throw new ResourceNotFoundException("Entity Author is empty");
26         }
27         return arepo.findAll();
28     }
29
30     @Override
31     public Author findById(Integer id) {
32         Optional<Author> author = arepo.findById(id);
33         if (author.isPresent()) {
34             return author.get();
35         }
36         throw new ResourceNotFoundException("Author with id: " + id + " is not present");
37     }
38
39     @Override
40     public Author create(Author entity) {
41         return arepo.save(entity);
42     }
43
44
45     @Override
46     public Author update(Integer id, Author entity) {
47         if (arepo.findById(id).isPresent()) {
48             entity.setId(id);
49             return arepo.save(entity);
50         }
51         throw new ResourceNotFoundException("Author with id: " + id + " is not present");
52     }
53
54     @Override

```

Сл.16 Класата AuthorService

```

50     }
51     throw new ResourceNotFoundException("Author with id: " + id + " is not present");
52 }
53
54 @Override
55 public void deleteById(Integer id) {
56     if (arepo.findById(id).isEmpty()) {
57         throw new ResourceNotFoundException("Author with id: " + id + " is not present");
58     }
59     arepo.deleteById(id);
60 }
61
62
63 public List<Author> findByFirstName(String firstName) {
64     List<Author> authorList = arepo.findByFirstName(firstName);
65     if (authorList.isEmpty()) {
66         throw new ResourceNotFoundException("Author with first name : " + firstName + " is not present");
67     }
68
69     return arepo.findByFirstName(firstName);
70 }
71
72 }
73

```

Сл.17 Класата AuthorService

Прво се внесува пакетот во кој се наоѓа класата и се внесуваат потребните анотации. Методот `findAll()`; прикажува листа на сите автори. Ако списокот е празен, фрла `ResourceNotFoundException`, што покажува дека нема автори во базата на податоци. Методот `findById(Integer id)` пребарува автор според неговото `id`. Доколку авторот постои го враќа, во спротивно, фрла `ResourceNotFoundException`. `Create(Author entity)` прифаќа објект автор и го зачувува во базата на податоци користејќи го методот за зачувување во складиштето. `Update(Integer id, Author entity)` ажурира информации за постоечки автор. Прво, проверува дали постои автор со наведеното ID. Ако е така, го поставува ID на предадениот ентитет и го зачувува. Ако не, фрла `ResourceNotFoundException`. `DeleteById(Integer id)` бриши автор според неговото ID. Слично на `findById`, проверува дали авторот постои пред да се обиде да го избрише. Ако авторот не постои тој фрла `ResourceNotFoundException`. `FindByFirstName(String firstName)` пребарува автори според нивното име. Доколку не се најдат автори, фрла `ResourceNotFoundException`.

Следно, ќе ја разгледаме класата BookService која го имплементира интерфејсот Service.

```
BookService.java x
1 package mk.fict.bookshelf.service.implementation;
2
3 import java.util.List;
4
12 @org.springframework.stereotype.Service
13 public class BookService implements Service<Book, Integer> {
14
15     @Autowired
16     private BookRepository brepo;
17
18     @Override
19     public List<Book> findAll() {
20         if (brepo.findAll().isEmpty()) {
21             throw new ResourceNotFoundException("Entity Book is empty");
22         }
23         return brepo.findAll();
24     }
25
26     @Override
27     public Book findById(Integer id) {
28         Optional<Book> book = brepo.findById(id);
29         if (brepo.findById(id).isPresent()) {
30             return book.get();
31         }
32         throw new ResourceNotFoundException("Book with id: " + id + " is not present");
33     }
34
35     @Override
36     public Book create(Book entity) {
37         return brepo.save(entity);
38     }
39
40     @Override
41     public Book update(Integer id, Book entity) {
42         if (brepo.findById(id).isPresent()) {
43             entity.setId(id);
44             return brepo.save(entity);
45         }
46         throw new ResourceNotFoundException("Book with id: " + id + " is not present");
47     }
48
49     }
50
51 }
```

Сл.18 Класата BookService

```

49         throw new ResourceNotFoundException( "Book with id: " + id + " is not present" );
50     }
51 }
52
53 @Override
54 public void deleteById(Integer id) {
55     if (brepo.findById(id).isEmpty()) {
56         throw new ResourceNotFoundException("Book with id: " + id + " is not present");
57     }
58     brepo.deleteById(id);
59 }
60 }
61
62 public Book findByIsbn(String isbn) {
63     return brepo.findByIsbn(isbn);
64 }
65 }
66 }
67 }
68 }

```

Сл.19 Класата BookService

Оваа класа го имплементира Service интерфесот. Прво, го внесуваме пакетот во кој се наоѓа класата и ги внесуваме потребни анотации. Методот `findAll()` ги презема сите книги од базата на податоци. Доколку не се пронајдени книги, фрла `ResourceNotFoundException`, што покажува дека колекцијата е празна. `findById(Integer id)` пребарува книги според ID. Проверува дали книгата постои во базата. Ако се најде, ја враќа книгата. Доколку не се најде, фрла `ResourceNotFoundException`. `Create(Book entity)` зачувува нова книга во базата користејќи го `save` методот од базата. `Update(Integer id, Book entity)` ажурира информации за постоечка книга. Прво проверува дали книгата постои. Доколку е така, го поставува соодветното ID и зачувува. Доколку книгата не постои, фрла `ResourceNotFoundException`. `DeleteById(Integer id)` бриши книга според дадено ID. Доколку книгата не постои враќа `ResourceNotFoundException`. `FindByIsbn(String isbn)` презема книга според неговиот ISBN.

Во пакетот `mk.fict.bookshelf.api` се наоѓаат повеќе класи каде се креираат апи за соодветните endpoints.

```
22 @CrossOrigin
23 @RestController
24 @RequestMapping(Endpoints.AUTHORS)
25 public class AuthorController {
26
27     @Autowired
28     AuthorService as;
29
30     @GetMapping
31     public List<Author> findAll() {
32         return as.findAll();
33     }
34
35     @GetMapping("/{id}")
36     public Author findById(@PathVariable(value = "id") Integer id) {
37         return as.findById(id);
38     }
39
40     @PostMapping
41     @ResponseStatus(value = HttpStatus.CREATED)
42     public Author create(@RequestBody Author entity) {
43
44         return as.create(entity);
45     }
46
47
48     @PutMapping("/{id}")
49     @ResponseStatus(value = HttpStatus.OK)
50     public Author update(@PathVariable(value = "id", required = true) Integer id, @RequestBody Author entity) {
51         return as.update(id, entity);
52     }
53
54     @DeleteMapping("/{id}")
55     @ResponseStatus(value = HttpStatus.NO_CONTENT)
56     public void deleteById(@PathVariable(value = "id", required = true) Integer id) {
57         as.deleteById(id);
58     }
59
60     @GetMapping("/firstname/{firstName}")
61     public List<Author> findByFirstName(@PathVariable(value = "firstName") String firstName) {
62         return as.findByFirstName(firstName);
63     }
64 }
```

Сл.20 AuthorController класа

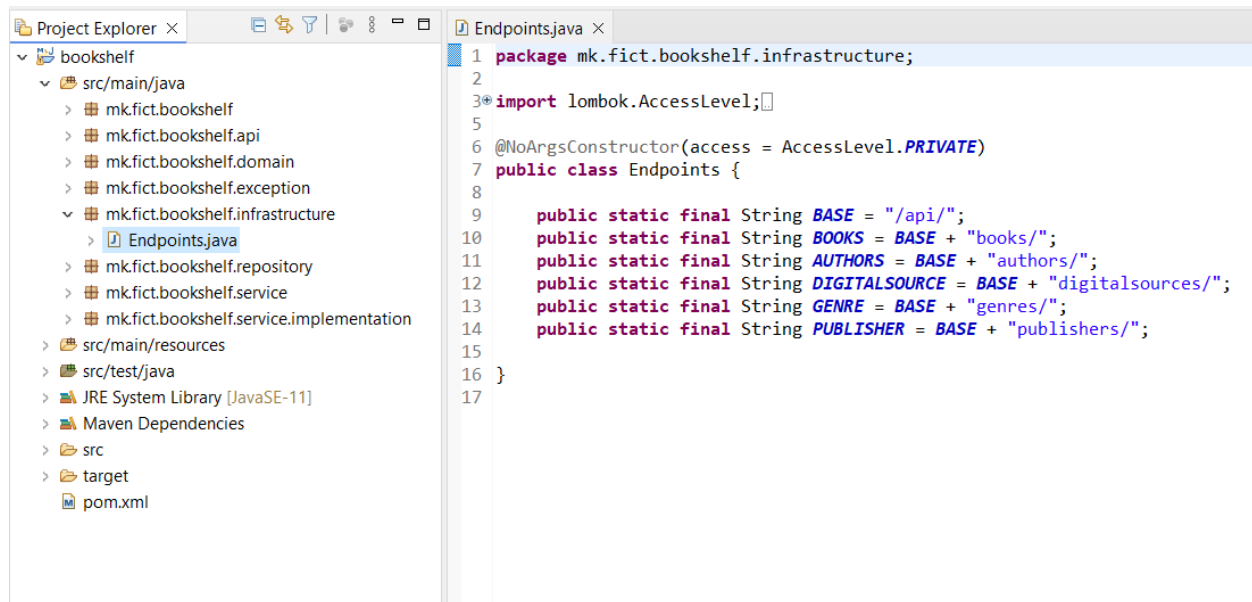
Прво се внесува името на пакетот и соодветните анотации. Во оваа класата претставени се следните API endpoints: `@GetMapping` – за сите автори, каде патеката ќе биде `/authors`. Враќа листа на автори повикувајќи го `findAll()` методот од `AuthorService`. `@GetMapping` - `/authors/{id}` дава баран автор според внесено ID. `@PostMapping` - `/authors` креира нов автор. Во `@RequestBody` се внесуваа соодветниот JSON request body за ентитетот автор. Доколку успешно се креира

авторот, враќа статус 201 Created. @PutMapping - /authors/{id} ажурира автор. Во @RequestBody ги внесува промените за ентитетот автор и го ажурира авторот со специфичен ID. Доколку е во ред се добива статус 200 OK. @DeleteMapping - /authors/{id} брише автор според некое ID. Се враќа статус 204 No Content. @GetMapping - /authors/firstname/{firstname} добива листа на автори кои го имаат соодветното име.

```
BookController.java ×
22 @CrossOrigin
23 @RestController
24 @RequestMapping(Endpoints.BOOKS)
25 public class BookController {
26
27     @Autowired
28     private BookService bs;
29
30     @GetMapping
31     public List<Book> findAll() {
32         return bs.findAll();
33     }
34
35     @GetMapping("/{id}")
36     public Book findById(@PathVariable(value = "id") Integer id) {
37         return bs.findById(id);
38     }
39
40     @PostMapping
41     @ResponseStatus(value = HttpStatus.CREATED)
42     public Book create(@RequestBody Book entity) {
43         return bs.create(entity);
44     }
45
46
47     @PutMapping("/{id}")
48     @ResponseStatus(value = HttpStatus.OK)
49     public Book update(@PathVariable(value = "id", required = true) Integer id, @RequestBody Book entity) {
50         return bs.update(id, entity);
51     }
52
53     @DeleteMapping("/{id}")
54     @ResponseStatus(value = HttpStatus.NO_CONTENT)
55     public void deleteById(@PathVariable(value = "id", required = true) Integer id) {
56         bs.deleteById(id);
57     }
58
59     @GetMapping("/isbn/{isbn}")
60     public Book findByIsbn(@PathVariable(value = "isbn") String isbn) {
61         return bs.findByIsbn(isbn);
62     }
63
64 }
```

Сл.21 BookController класа

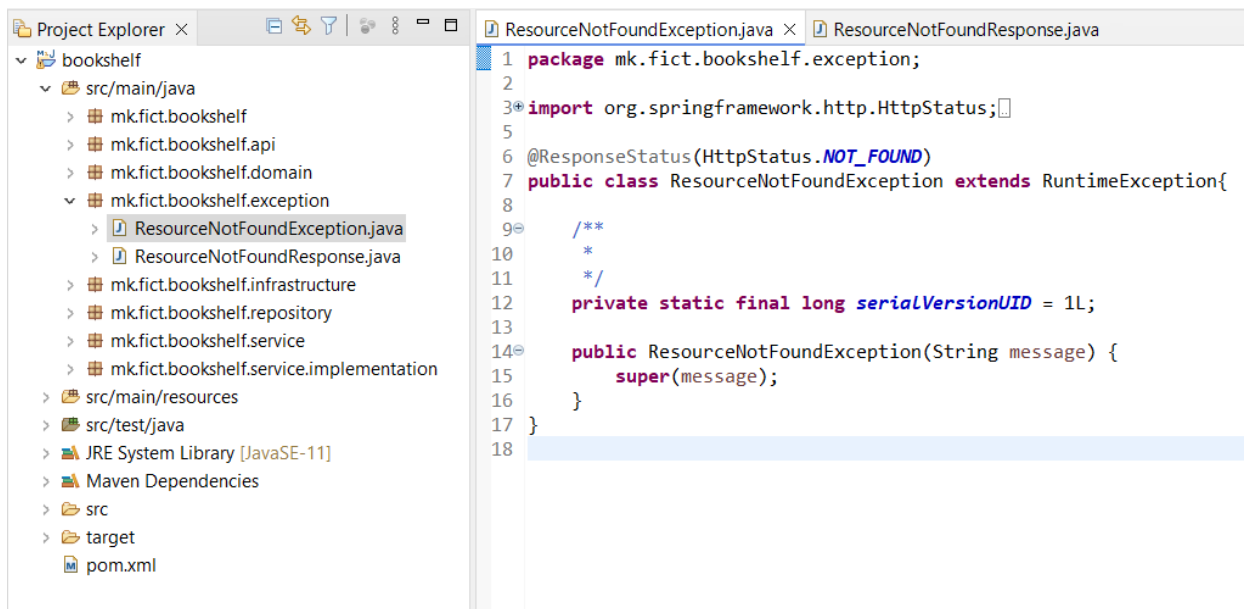
Класата BookController дефинира RESTful API за управување со ентитети во Spring апликација. Како и во погорната класа, потребно е да се внесе името на пакетот и потребните анотации. После тоа, се креираат endpoints за get /books, /books/{id}, post /books, put /books/{id}, delete /books/{id}, get /books/isbn/{isbn}.



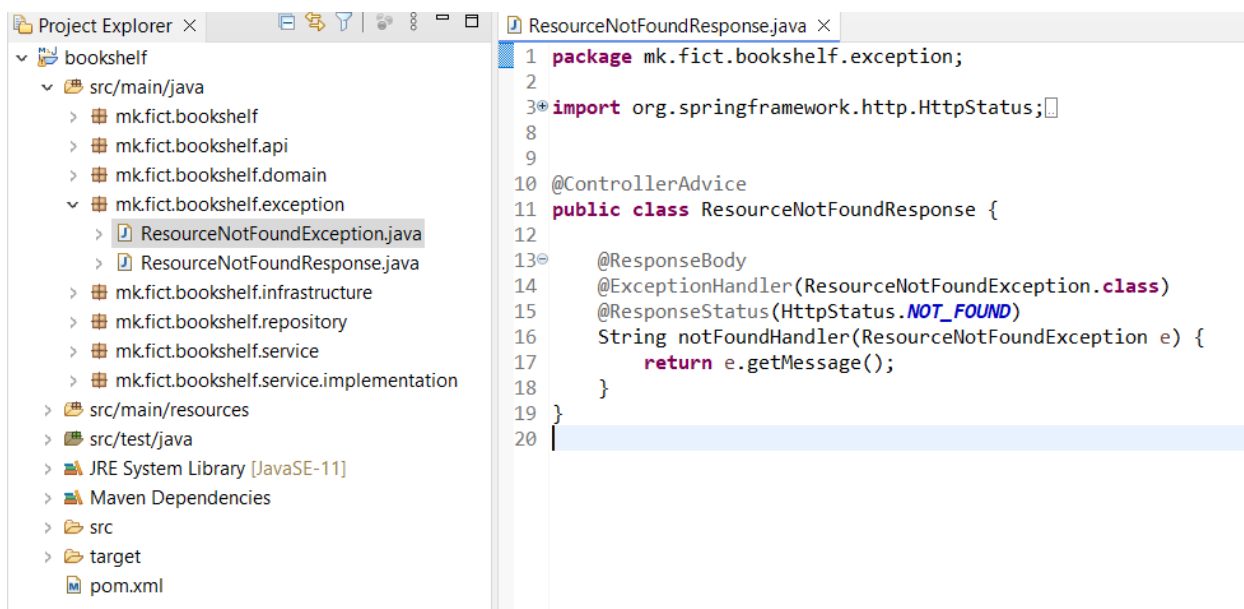
Сл.22 Класа Endpoints

Во класата Endpoints во пакетот mk.fict.bookshelf.infrastructure се дефинираат патеките – API endpoint paths.

Во пакетот mk.fict.bookshelf.exception се наоѓаат две класи кој враќаат Exception кога тоа е потребно.



Сл.23 ResourceNotFoundException



Сл.24 ResourceNotFoundResponse

Во pom.xml фајлот се опфатени основните библиотеки за развој на web апликација која е во интеракција со базата на податоци – PostgreSQL. Овој xml код претставува Maven POM(Project Object Model) за нашата Spring Boot апликација – bookshelf.

```

bookshelf/pom.xml x
https://maven.apache.org/xsd/maven-4.0.0.xsd (xsi:schemaLocation)
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"
5   <modelVersion>4.0.0</modelVersion>
6   <parent>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-parent</artifactId>
9     <version>2.4.3</version>
10    <relativePath /> <!-- lookup parent from repository -->
11  </parent>
12  <groupId>mk.fict</groupId>
13  <artifactId>bookshelf</artifactId>
14  <version>0.0.1-SNAPSHOT</version>
15  <name>bookshelf</name>
16  <description>bookshelf</description>
17  <properties>
18    <java.version>11</java.version>
19  </properties>
20  <dependencies>
21    <dependency>
22      <groupId>org.springframework.boot</groupId>
23      <artifactId>spring-boot-starter-data-jpa</artifactId>
24    </dependency>
25    <dependency>
26      <groupId>org.springframework.boot</groupId>
27      <artifactId>spring-boot-starter-web</artifactId>
28    </dependency>
29
30    <dependency>
31      <groupId>org.postgresql</groupId>
32      <artifactId>postgresql</artifactId>
33      <scope>runtime</scope>
34    </dependency>
35    <dependency>
36      <groupId>org.projectlombok</groupId>
37      <artifactId>lombok</artifactId>
38      <optional>true</optional>
39    </dependency>
40    <dependency>
41      <groupId>org.springframework.boot</groupId>

```

Сл.25 pom.xml

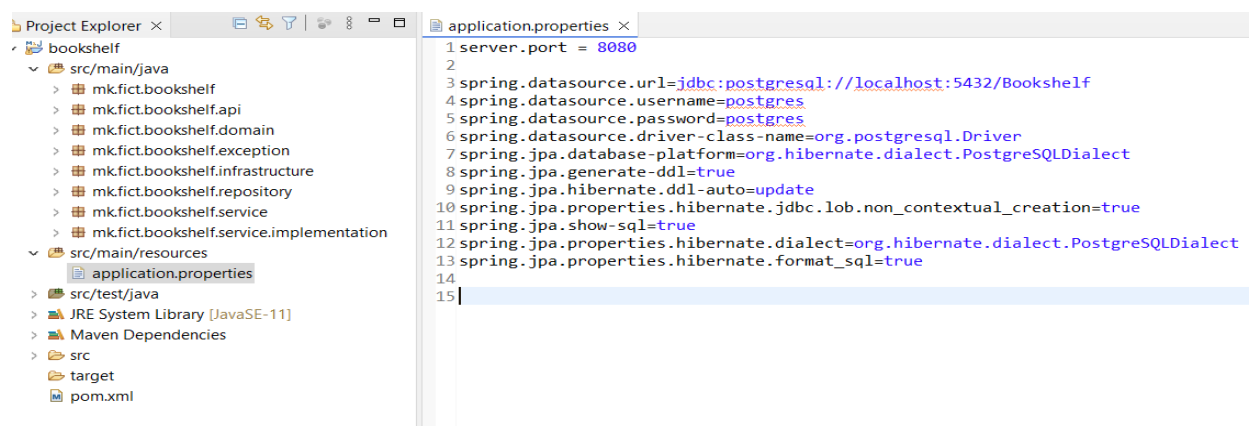
```

40    </dependency>
41    <groupId>org.springframework.boot</groupId>
42    <artifactId>spring-boot-starter-test</artifactId>
43    <scope>test</scope>
44  </dependency>
45  <dependency>
46    <groupId>io.springfox</groupId>
47    <artifactId>springfox-boot-starter</artifactId>
48    <version>3.0.0</version>
49  </dependency>
50
51
52 </dependencies>
53
54 <build>
55   <plugins>
56     <plugin>
57       <groupId>org.springframework.boot</groupId>
58       <artifactId>spring-boot-maven-plugin</artifactId>
59     <configuration>
60       <excludes>
61         <exclude>
62           <groupId>org.projectlombok</groupId>
63           <artifactId>lombok</artifactId>
64         </exclude>
65       </excludes>
66     </configuration>
67   </plugin>
68 </plugins>
69 </build>
70
71 </project>
72 |

```

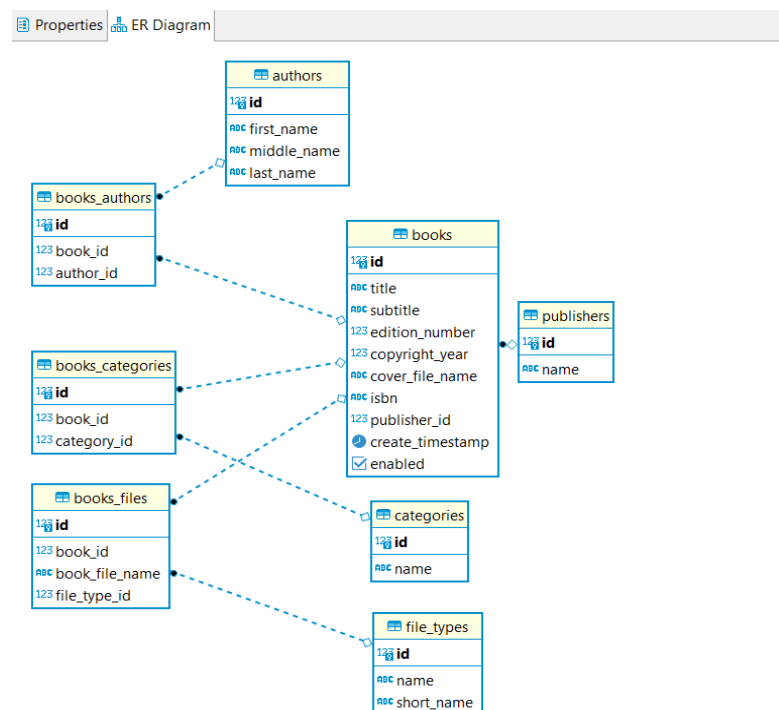
Сл.26 pom.xml

Во пакетот src/main/resources се наоѓа фајлот application.properties кој претставува конфигурација за поврзување на Spring Boot апликацијата со базата на податоци PostgreSQL, специфицирајќи ги деталите за врската, однесувањето на Hibernate за управување со шемата и овозможувајќи SQL логирање за цели на debugging.



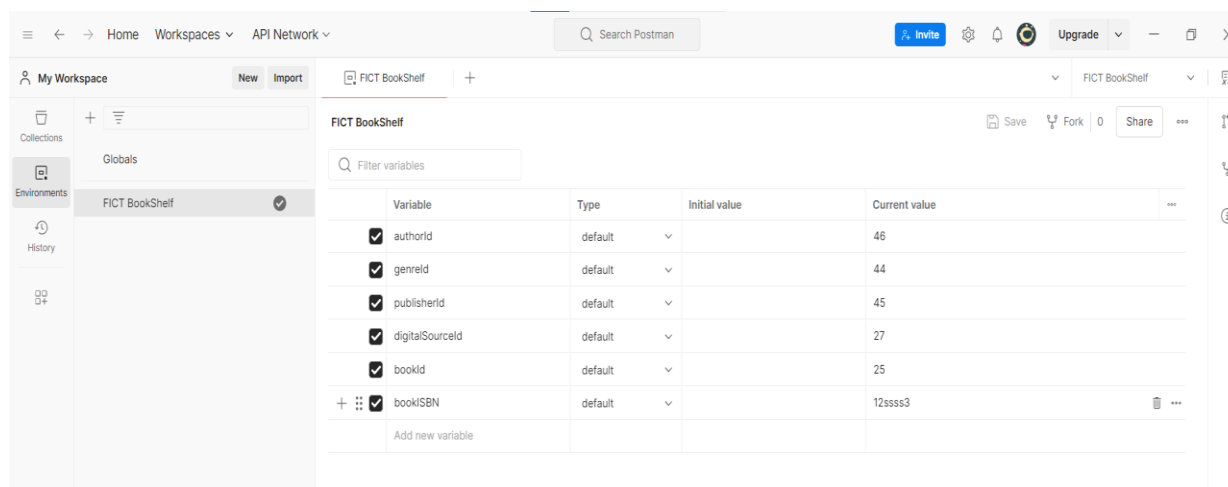
```
1 server.port = 8080
2
3 spring.datasource.url=jdbc:postgresql://localhost:5432/Bookshelf
4 spring.datasource.username=postgres
5 spring.datasource.password=postgres
6 spring.datasource.driver-class-name=org.postgresql.Driver
7 spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
8 spring.jpa.generate-ddl=true
9 spring.jpa.hibernate.ddl-auto=update
10 spring.jpa.properties.hibernate.jdbc.lob.non_contextual_creation=true
11 spring.jpa.show-sql=true
12 spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
13 spring.jpa.properties.hibernate.format_sql=true
14
15
```

Сл.27 application.properties



Сл.28 ER Дијаграмот на базата на податоци bookshelf

Во овој пример, автоматското тестирање ќе биде извршено во Postman. Прв чекор е креирање на Environment.



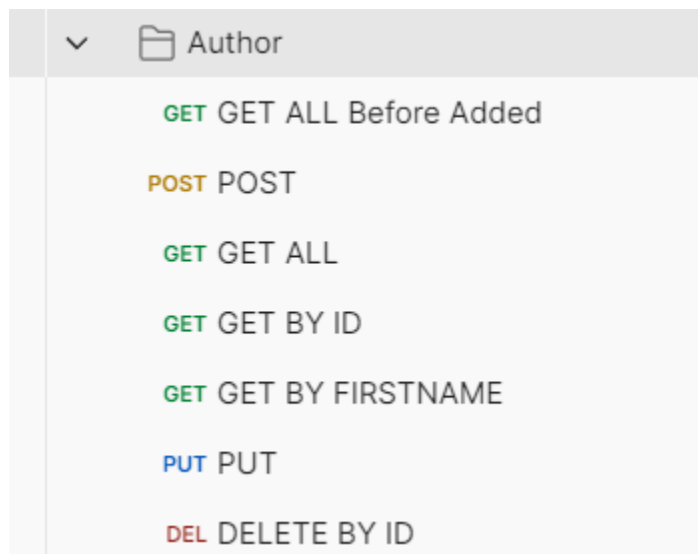
Сл.29 FICT BookShelf environment

Во оваа околина внесени се потребните варијабли, како и статички вредности за истите.

На сликата подолу е прикажана колекцијата која ќе се користи. Заради заштеда на простор, во оваа семинарска работа ќе ги прикажеме тестовите само за Author и Book.

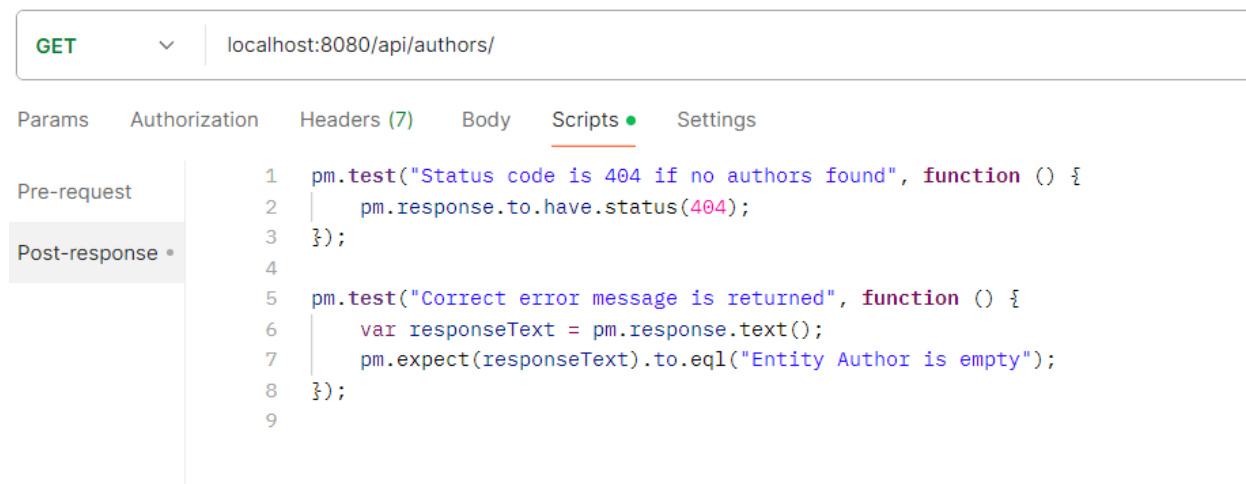


Сл.30 Колекцијата BookShelf FICT



Сл.31 Колекција Author

Првиот тест ќе биде за endpoint `/api/authors/`.

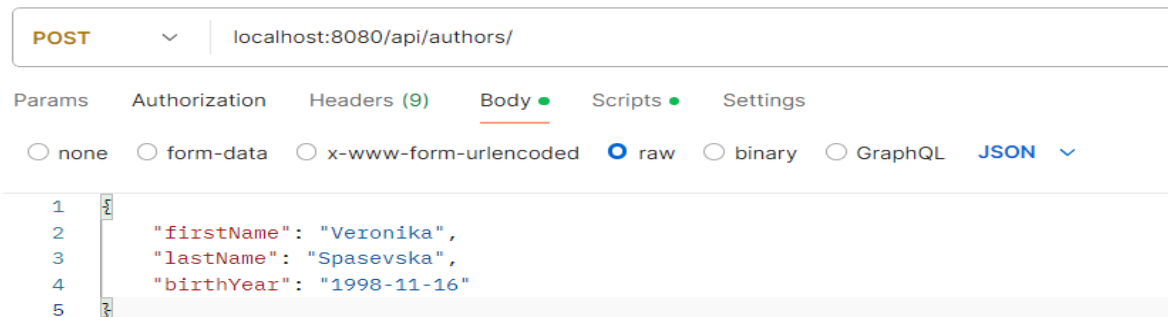


Сл.32 Get All Before Added

Првиот код проверува дали HTTP response статус кодот е 404, што означува дека бараниот ресурс(автор) не е пронајден. Вториот код потврдува дека response body

ја содржи точната порака за грешка, осигурувајќи дека се совпаѓа со очекуваниот текст: "Entity Author is empty".

Следен е методот POST Author, каде во body го внесуваме авторот кој сакаме да го запишеме.



Сл.33 POST authors

Во овој случај, test скриптите се следните:

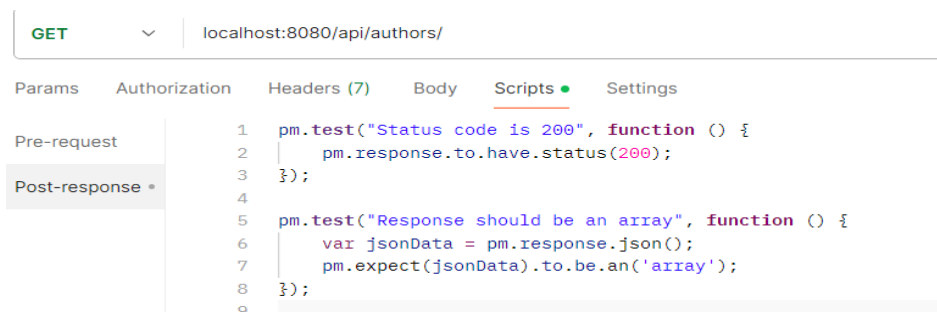
```
1  pm.test("Status code is 201", function () {
2    pm.response.to.have.status(201);
3  });
4
5  pm.test("Response should contain the created author", function () {
6    var jsonData = pm.response.json();
7    pm.expect(jsonData).to.have.property('firstName');
8    pm.expect(jsonData).to.have.property('lastName');
9    pm.expect(jsonData).to.have.property('birthYear');
10  });
11 pm.test("Response contains an ID", function () {
12   var jsonData = pm.response.json();
13   pm.expect(jsonData).to.have.property('id');
14   pm.environment.set("authorId", jsonData.id); // Store the "id" in an environment
        variable
15  });
16
```

Сл.34 Тест скрипти за post author

Првиот тест потврдува дека HTTP response статус кодот е 201, што покажува дека авторот е успешно креиран. Вториот тест проверува дали response содржи

properties за firstName, lastName и birthYear, потврдувајќи дека е вратена точната структура на податоци за креираниот автор. Третиот тест осигурува дека одговорот вклучува својство id, кое уникатно го идентификува креираниот автор. Исто така, го складира тоа id во environment variable наречена authorId за употреба во следните барања.

Следен request е get all со endpoint /api/authors.



Сл.35 Тест скрипти за get all

Првиот тест потврдува дека HTTP response status code е 200, означувајќи успешно барање. Вториот тест проверува дали response body е низа, потврдувајќи дека API враќа низа од автори.

Тестот за get by id е прикажан на сликата подолу:



```
GET localhost:8080/api/authors/{{authorId}}

Params Authorization Headers (7) Body Scripts Settings


Pre-request
Post-response *

1 pm.test("Status code is 200", function () {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test("Author should have the required fields", function () {
6   var jsonData = pm.response.json();
7   pm.expect(jsonData).to.have.property('id');
8   pm.expect(jsonData).to.have.property('firstName');
9   pm.expect(jsonData).to.have.property('lastName');
10  pm.expect(jsonData).to.have.property('birthYear');
11 });
12
13 pm.test("Correct ID is returned", function () {
14   var jsonData = pm.response.json();
15   pm.expect(jsonData.id).to.eql(Number(pm.variables.get("authorId"))); // Convert the
    variable to a number
16 });
```

Сл.36 Тест скрипти за get by id

Првиот тест потврдува дека одговорот на API враќа статус код 200. Вториот тест се осигурува дека JSON response ги вклучува потребните полиња(id, firstName, lastName, birthYear). Третиот тест потврдува дека id-то вратено во response се совпаѓа со променливата authorId.

Следени се тест сценаријата за get by firstName.



```
GET localhost:8080/api/authors/firstName/Veronika

Params Authorization Headers (7) Body Scripts Settings

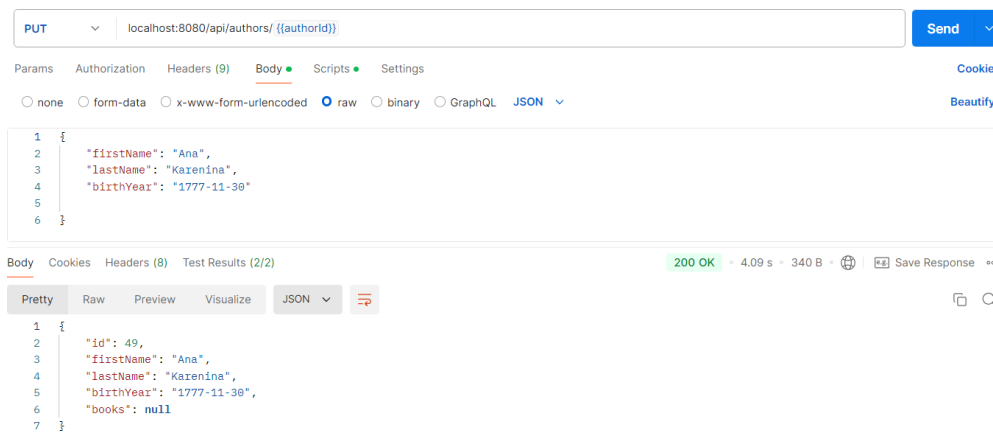
Pre-request
Post-response *

1 pm.test("Status code is 200", function () {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test("Response is an array", function () {
6   var jsonData = pm.response.json();
7   pm.expect(jsonData).to.be.an('array');
8 });
9
10 pm.test("All authors have the firstName 'Veronika'", function () {
11   var jsonData = pm.response.json();
12
13   jsonData.forEach(function (author) {
14     pm.expect(author.firstName).to.eql("Veronika");
15   });
16 });
17
```

Сл.37 Тест скрипта за get by firstName

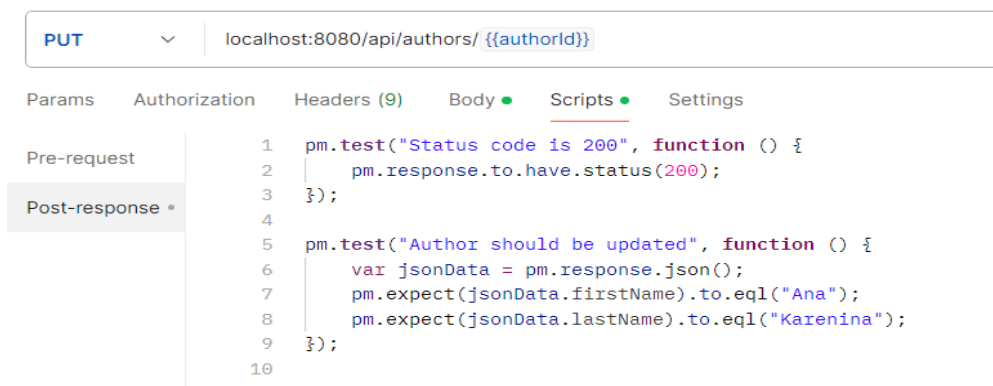
Првиот тест проверува дали статусот на response е 200, што укажува на успешно барање. Вториот тест осигурува дека одговорот е низа, што е корисно за API кои треба да враќаат низи. Третиот тест поминува низ секој автор во низата добиена на response за да потврди дека сите автори имаат firstName поставено да биде Veronika.

На сликата подолу е прикажан PUT requestot каде во request body внесуваме автор во JSON.



Сл.38 PUT author

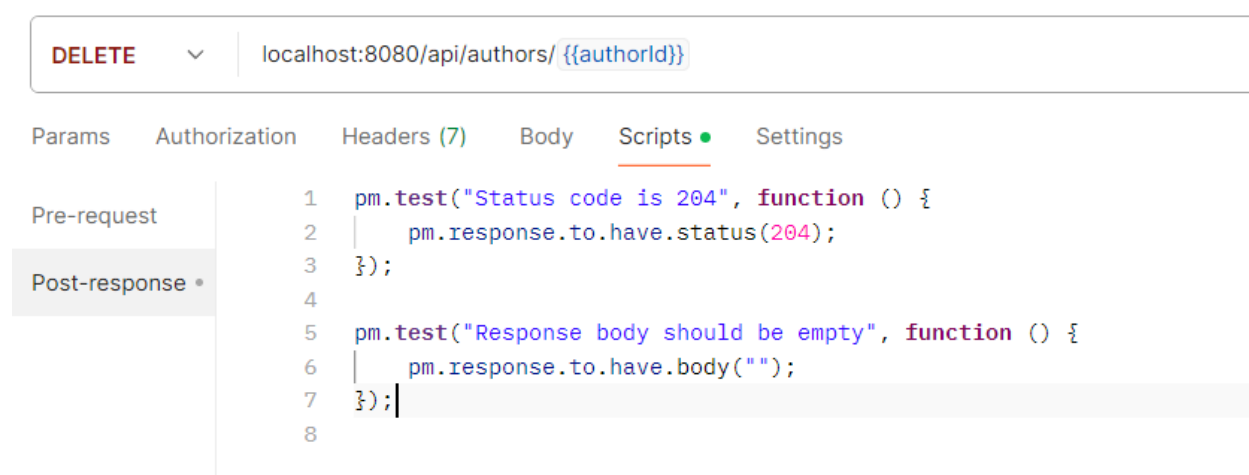
На следната слика се прикажани тест сценаријата за овој метод.



Сл.39 Тест скрипти за put author

Првиот тест проверува дали HTTP response status code е 200. Вториот тест потврдува дека името на ажурираниот автор е “Ана” и презимето “Karenina”.

Следен request е delete by id, каде првиот тест потврдува дека response status code е 204, што означува успешен request без содржина. Вториот тест осигурува дека response body е празно.



Сл.40 Тест скрипти за delete by id

Следно, ќе ги разгледаме тест скриптите и requests за ентитетот Book. За да се креира книга, мора претходно да ги имаме следните ентитети во базата на податоци: Author, Publisher и Genre. Од таа причина, креирана е следната скрипта:

Overview	Authorization	Scripts ●
Pre-request *		<pre> 1 // Send request for creating a Genre and store genreId 2 pm.sendRequest({ 3 url: 'http://localhost:8080/api/genres/', 4 method: 'POST', 5 header: { 6 'Content-Type': 'application/json' 7 }, 8 body: { 9 mode: 'raw', 10 raw: JSON.stringify({ 11 "name": "Drama" 12 }) 13 } 14 }, function (err, res) { 15 if (err) { 16 console.log(err); 17 } else { 18 var jsonData = res.json(); 19 if (jsonData && jsonData.id) { 20 pm.environment.set("genreId", jsonData.id); // Store the genreId in 21 // environment variable 22 console.log("genreId has been set to: " + jsonData.id); 23 } 24 } 25 }); 26 27 // Send request for creating a Publisher and store publisherId 28 pm.sendRequest({ 29 url: 'http://localhost:8080/api/publishers/', // Use the complete URL 30 method: 'POST', 31 header: { 32 'Content-Type': 'application/json' </pre>
Post-response		

Сл.41 Pre-request скрипта за Book

Overview	Authorization	Scripts ●
Pre-request *		<pre> 30 header: { 31 'Content-Type': 'application/json' 32 }, 33 body: { 34 mode: 'raw', 35 raw: JSON.stringify({ 36 "name": "Makedonska Kukja" 37 }) 38 } 39 }, function (err, res) { 40 if (err) { 41 console.log(err); 42 } else { 43 var jsonData = res.json(); 44 if (jsonData && jsonData.id) { 45 pm.environment.set("publisherId", jsonData.id); // Store the publisherId in 46 // environment variable 47 console.log("publisherId has been set to: " + jsonData.id); 48 } 49 } 50 }); 51 52 // Send request for creating an Author and store authorId 53 pm.sendRequest({ 54 url: 'http://localhost:8080/api/authors/', // Use the complete URL 55 method: 'POST', 56 header: { 57 'Content-Type': 'application/json' 58 }, 59 body: { 60 mode: 'raw', 61 raw: JSON.stringify({ 62 "firstName": "Veronika", 63 "lastName": "Kukja" 64 }) 65 } 66 }, function (err, res) { 67 if (err) { 68 console.log(err); 69 } else { 70 var jsonData = res.json(); 71 if (jsonData && jsonData.id) { 72 pm.environment.set("authorId", jsonData.id); // Store the authorId in 73 // environment variable 74 console.log("authorId has been set to: " + jsonData.id); 75 } 76 } 77 }); </pre>
Post-response		

Сл.42 Pre-request скрипта за Book

```

51 // Send request for creating an Author and store authorId
52 pm.sendRequest({
53   url: 'http://localhost:8080/api/authors/', // Use the complete URL
54   method: 'POST',
55   header: {
56     'Content-Type': 'application/json'
57   },
58   body: {
59     mode: 'raw',
60     raw: JSON.stringify({
61       "firstName": "Veronika",
62       "lastName": "Spasevska",
63       "birthYear": "1977-11-30"
64     })
65   }
66 }, function (err, res) {
67   if (err) {
68     console.log(err);
69   } else {
70     var jsonData = res.json();
71     if (jsonData && jsonData.id) {
72       pm.environment.set("authorId", jsonData.id); // Store the authorId in
73       // environment variable
74       console.log("authorId has been set to: " + jsonData.id);
75     }
76   }
77 });

```

Сл.43 Pre-request скрипта за Book

Првиот request е POST book, каде во request body внесуваме книга во соодветен JSON формат.

The screenshot shows the Postman interface for a POST request to `localhost:8080/api/books/`. The request body is in JSON format, containing the following structure:

```

{
  "title": "TestBook",
  "rating": 1,
  "isbn": "12ssss3",
  "publishedDate": "2020-12-11T23:00:00.000+00:00",
  "genres": [
    {
      "id": "{{genreId}}"
    }
  ],
  "publishers": {
    "id": "{{publisherId}}"
  },
  "authors": [
    {
      "id": "{{authorId}}"
    }
  ]
}

```

The status bar at the bottom indicates a **201 Created** response, with a duration of 319 ms and a body size of 502 B. The interface also shows tabs for Params, Authorization, Headers (9), Body, Scripts, and Settings, along with a Beautify button.

Сл.44 POST book

POST localhost:8080/api/books/Params Authorization Headers (9) Body Scripts Settings

Pre-request

Post-response •

```
1 pm.test("Status code is 201", function () {
2   pm.response.to.have.status(201);
3 });
4
5 pm.test("Response contains the created book with ID", function () {
6   var jsonData = pm.response.json();
7   pm.expect(jsonData).to.have.property('id');
8   pm.expect(jsonData).to.have.property('title');
9   pm.expect(jsonData).to.have.property('rating');
10  pm.expect(jsonData).to.have.property('isbn');
11  pm.expect(jsonData).to.have.property('publishedDate');
12  pm.expect(jsonData).to.have.property('genres');
13  pm.expect(jsonData).to.have.property('authors');
14 });
15
16 pm.test("Store book ID in environment variable", function () {
17   var jsonData = pm.response.json();
18   pm.environment.set("bookId", jsonData.id); // Store the "id" of the new book in
    environment variable
19 });
20 pm.test("Store ISBN in environment variable", function () {
21   var jsonData = pm.response.json();
22   pm.environment.set("bookISBN", jsonData.isbn); // Store the "id" of the new book in
    environment variable
23 });
```

Сл.45 Скрипти за POST book

Првиот тест потврдува дека response API враќа 201 статус код. Втората скрипта осигурува дека одговорот ги содржи сите потребни својства за креираната книга. Третиот тест код го складира новосоздадениот id во environment variable за подоцнежна употреба. Четвртиот тест го складира ISBN во environment variable.

Следен request е get all books.

GET

localhost:8080/api/books/

Params

Authorization

Headers (7)

Body

Scripts ●

Settings

Pre-request

Post-response •

```
1  pm.test("Status code is 200", function () {
2    |    pm.response.to.have.status(200);
3  });
4
5  pm.test("Response should be an array", function () {
6    |    var jsonData = pm.response.json();
7    |    pm.expect(jsonData).to.be.an('array');
8  });
9
10 pm.test("Each book should have the required fields", function () {
11   |    var jsonData = pm.response.json();
12   |    jsonData.forEach(function (book) {
13   |      |    pm.expect(book).to.have.property('id');
14   |      |    pm.expect(book).to.have.property('title');
15   |      |    pm.expect(book).to.have.property('rating');
16   |      |    pm.expect(book).to.have.property('isbn');
17   |      |    pm.expect(book).to.have.property('publishedDate');
18   |      |    pm.expect(book).to.have.property('genres');
19   |      |    pm.expect(book).to.have.property('authors');
20   |    });
21  });
22
```

Сл.46 get all books

Првиот тест потврдува дека response статус кодот е 200, осигурувајќи дека барањето е успешно. Вториот тест проверува дали одговорот е низа, што е важно за правилна обработка на податоците. Третиот тест поминува низ сите објект на книги во низата за да се осигура дека ги содржи потребните својства: id, title, rating, isbn, publishedDate, genres и authors.

Следен request е get by id.

GET

localhost:8080/api/books/ {{bookId}}

Params

Authorization

Headers (7)

Body

Scripts

Settings

Pre-request

Post-response

```
1 pm.test("Status code is 200", function () {
2   pm.response.to.have.status(200);
3 });
4
5 pm.test("Book should have the required fields", function () {
6   var jsonData = pm.response.json();
7   pm.expect(jsonData).to.have.property('id');
8   pm.expect(jsonData).to.have.property('title');
9   pm.expect(jsonData).to.have.property('rating');
10  pm.expect(jsonData).to.have.property('isbn');
11  pm.expect(jsonData).to.have.property('publishedDate');
12  pm.expect(jsonData).to.have.property('genres');
13  pm.expect(jsonData).to.have.property('authors');
14 });
15
16 pm.test("Correct ID is returned", function () {
17   var jsonData = pm.response.json();
18   pm.expect(jsonData.id).to.eql(Number(pm.environment.get("bookId"))); // Assuming
    bookId is stored in environment
19 });
20
```

Сл.47 Get by id

Првиот тест потврдува дека response статус кодот е 200, што укажува на успешно барање. Вториот тест проверува дали вратениот објект за книга си содржи сите потребни својства. Третиот тест осигурува дека id-то на добиената книга се совпаѓа со bookId зачувано во environment variables, што е корисно за да се потврди дека се зема точната книга.

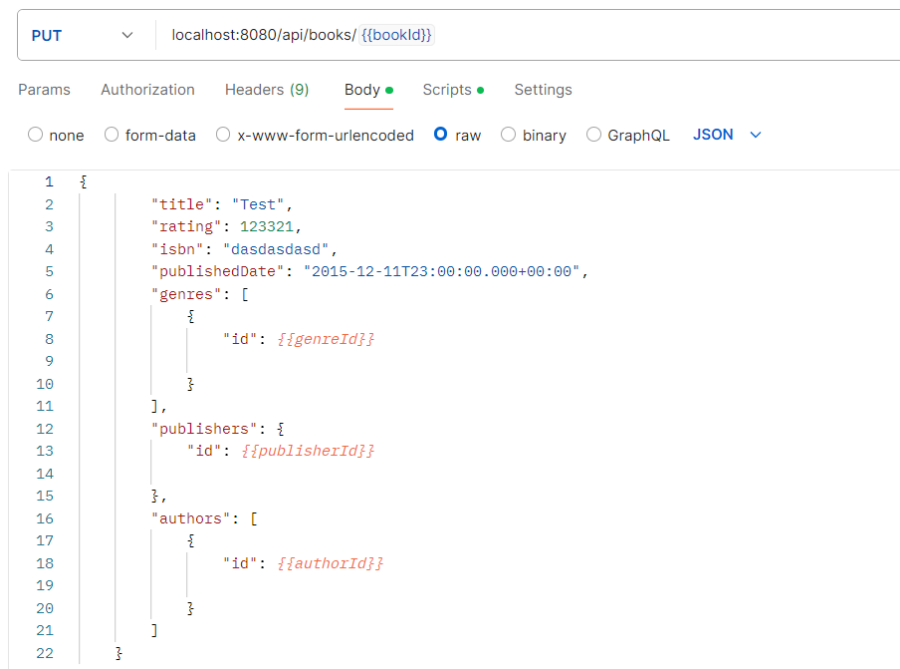
Следен request е get by isbn.



Сл.48 Get by isbn

Првиот тест осигурува дека response status code е 200. Вториот тест проверува дали isbn во одговорот се совпаѓа со bookISBN зачуван во environment variables.

Следен request е PUT book.



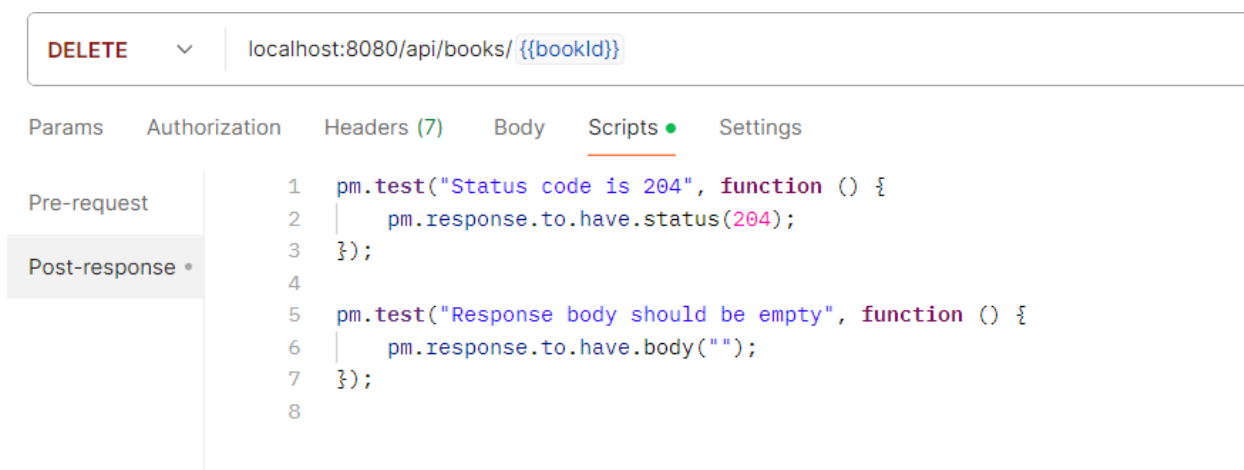
Сл.49 PUT book



Сл.50 Тест скрипти за put book

Првиот тест потврдува дека response status code е 200. Вториот код проверува дали насловот и ISBN на вратената книга се совпаѓаат со очекуваните ажурирани вредности.

Следен request е delete by id.

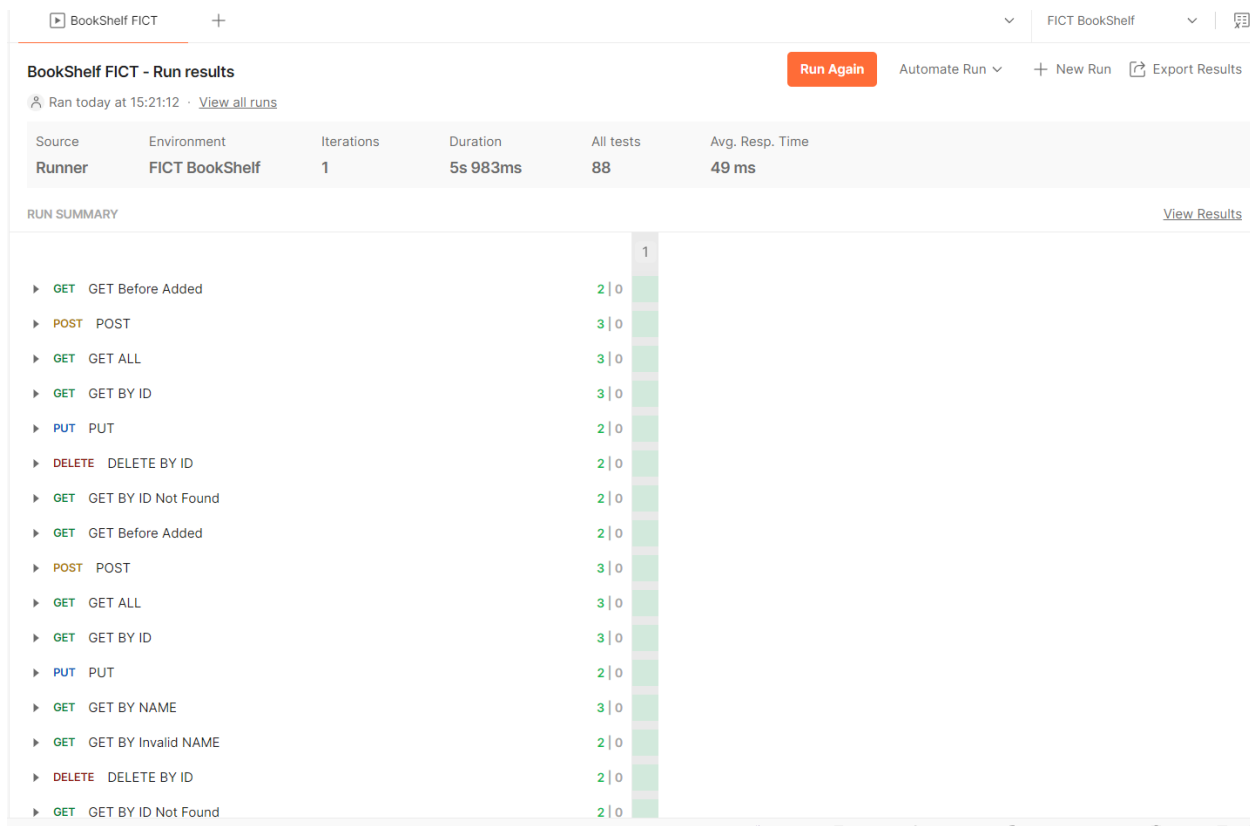


Сл.51 Delete by id

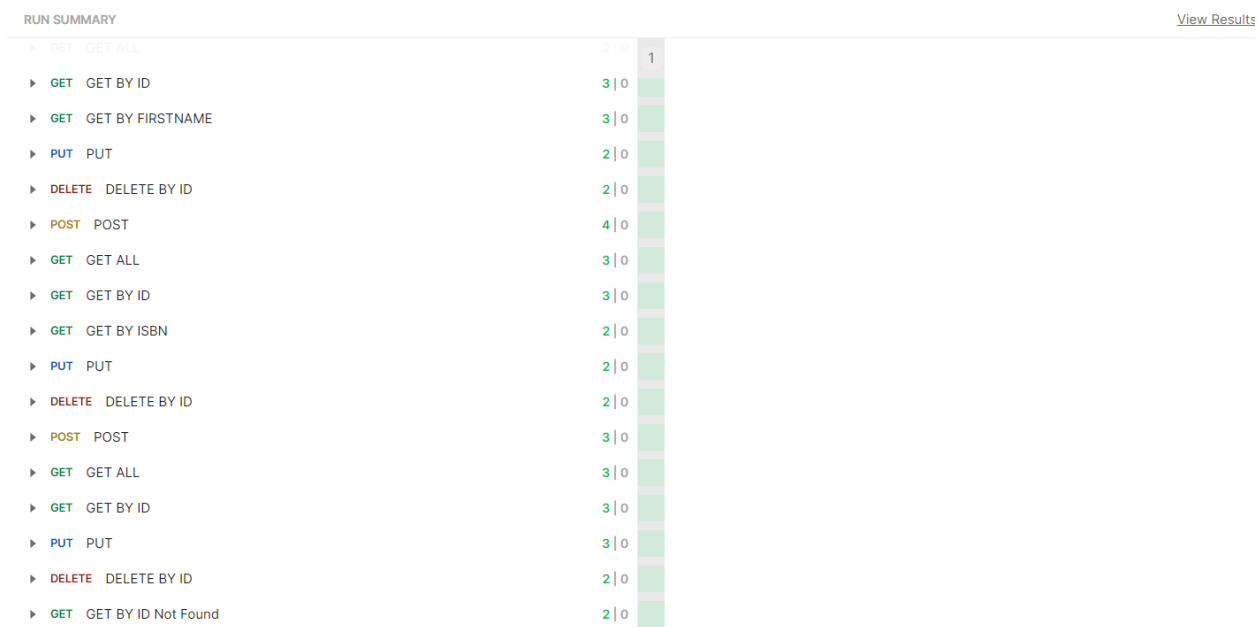
Првиот тест потврдува дека response status code е 204 што покажува дека барањето е успешно, но нема содржина за враќање. Вториот тест осигурува дека response body е празно, што се очекува за статусот 204.

The screenshot displays the Postman interface for running a collection. On the left, the 'Run order' panel lists 20 tests, each with a checkbox and a folder icon. The tests are: GET Before Added, POST, GET ALL, GET BY ID, PUT, DELETE BY ID, GET BY ID Not Found, GET Before Added, POST, GET ALL, GET BY ID, PUT, GET BY NAME, GET BY Invalid NAME, DELETE BY ID, GET BY ID Not Found, GET ALL Before Added, POST, and GET ALL. All tests are selected. On the right, the 'Run configuration' panel is visible. It has two tabs: 'Functional' (selected) and 'Performance'. Under 'Choose how to run your collection', 'Run manually' is selected. Under 'Run configuration', 'Iterations' is set to 1, 'Delay' is 0 ms, and 'Data file' is 'Select File'. There are checkboxes for 'Persist responses for a session' and 'Turn off logs during run', both of which are unchecked. An 'Advanced settings' link is present. At the bottom, there is an orange button labeled 'Run BookShelf FICT'.

Сл.52 Извршување на сите тестови во BookShelf колекцијата истовремено



Сл.53 Успешно извршување на сите тестови



Сл.54 Успешно извршување на сите тестови

All Tests		Passed (88)	Failed (0)	Skipped (0)	View Summary	
		PASS	Status code is 201			
		PASS	Response contains the created genre with ID			
		PASS	Store genre ID in environment variable			
GET GET ALL						
localhost:8080/api/genres/					200 OK	36 ms 280 B
		PASS	Status code is 200			
		PASS	Response should be an array			
		PASS	Each genre should have the required fields			
GET GET BY ID						
localhost:8080/api/genres/1					200 OK	42 ms 278 B
		PASS	Status code is 200			
		PASS	Genre should have the required fields			
		PASS	Correct ID is returned			
PUT PUT						
localhost:8080/api/genres/1					200 OK	33 ms 279 B
		PASS	Status code is 200			
		PASS	Genre should be updated			

Сл.55 Успешно извршување на сите тестови

Заклучок

Во оваа семинарска работа ги разгледавме клучните аспекти на софтверското тестирање, функционално тестирање, нефункционално тестирање како и рачно и мануелно тестирање. Акцентот беше ставен на придобивките и недостатоците на автоматизацијата, како и стратегиите за избор на тестови кои треба да се автоматизираат.

Во практичниот дел, го разгледавме процесот на креирање на Spring Boot апликација, поврзана со база на податоци PostgreSQL и креирање на автоматски тестови во Postman.

Тестирањето на софтвер е важно бидејќи обезбедува квалитет, брзо ги идентификува грешките, го подобрува задоволството на корисниците и ја подобрува безбедноста. Исто така, помага за оптимизирање на перформансите, обезбедува усогласеност со стандардите, ги ублажува ризиците и поттикнува континуирано подобрување. Тестирањето обезбедува сигурен производ кој ги задоволува потребите и очекувањата на корисниците.

Користена литература

- [1]Ammann, P. and Offutt, J., 2017. *Introduction to software testing*. Cambridge University Press
- [2]Singh, S.K. and Singh, A., 2012. *Software testing*. Vandana Publications
- [3]Dustin, E., Rashka, J. and Paul, J., 1999. *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional
- [4]Sneha, K. and Malle, G.M., 2017, August. Research on software testing techniques and software automation testing tools. In *2017 international conference on energy, communication, data analytics and soft computing (ICECDS)* (pp. 77-81). IEEE
- [5]Westerveld, D., 2021. *API Testing and Development with Postman: A practical guide to creating, testing, and managing APIs for automated software testing*. Packt Publishing Ltd
- [6]Kore, P.P., Lohar, M.J., Surve, M.T. and Jadhav, S., 2022. API Testing Using Postman Tool. *International Journal for Research in Applied Science and Engineering Technology*, 10(12), pp.841-43.