Dominic M. KRISTÁLY

PROGRAMARE WEB

Îndrumar de laborator

Editura
Universității
Transilvania
din Brașov

2026

EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV

Adresa:	Str. Iuliu Maniu nr. 41A
	500091 Brașov
Tel.:	0268 476 050
Fax:	0268 476 051
E-mail:	editura@unitbv.ro

Editură recunoscută CNCSIS, cod 81

ISBN 978-606-19-1888-1 (e-book)

Copyright © Autorul, 2026

Lucrarea a fost avizată de Consiliul Departamentului de Automatică și Tehnologia Informației, Facultatea de Inginerie electrică și știința calculatoarelor a Universității Transilvania din Brașov.

Cuprins

Laborator 1: Construirea unui API REST cu Spring Boot 5
1. Spring Framework, Spring Boot și Spring Initializr 5
2. Obiectivul laboratorului 7
3. Aplicația care va fi realizată 8
4. Pornirea proiectului 9
5. Punctul de plecare: un controller foarte simplu 10
6. Trecerea de la text la obiecte JSON 11
7. Returnarea unei liste de obiecte 13
8. Introducerea unei colecții persistente în memorie 14
9. Lucrul cu identificatori din URL 15
10. Controlul răspunsului HTTP cu ResponseEntity 15
11. Primirea datelor de la client: @RequestBody 16
12. Alegerea codului de stare potrivit la creare 17
13. Actualizarea unei resurse cu @PutMapping 18
14. Ștergerea unei resurse cu @DeleteMapping 19
15. Varianta completă: tot codul în controller 19
16. Ce face @RequestMapping 21
17. Testarea pas cu pas 21
18. Primele limite ale abordării 23
19. Separarea pe straturi 23
20. Ce se câștigă prin separare 27
21. O nouă problemă: obiectul expus la exterior coincide cu obiectul intern 27
22. Obiecte DTO 28
23. Primele idei de DDD 31
24. Întrebări recapitulative 32
25. Exerciții propuse 33
Laborator 2: Persistența datelor cu Spring Data JPA și MySQL în aplicații REST Spring Boot 34
1. Obiectivul laboratorului 34
2. Ce este Spring Data și ce problemă rezolvă 34
3. Aplicația care va fi realizată 47
4. Ce se schimbă față de laboratorul anterior 47
5. Dependențele necesare 47
6. Baza de date: două variante de lucru 48
7. Configurarea aplicației Spring Boot 49
8. Structura proiectului 52
9. Transformarea clasei Product într-o entitate persistentă 52
10. Introducerea repository-ului 54
11. Adaptarea DTO-ului de creare 55
12. Refactorizarea service-ului pentru baza de date 56
13. Controller-ul adaptat 57
14. Clasa principală 59
15. Ce se întâmplă la pornirea aplicației 59

16. Testarea aplicației ... 59
17. Popularea inițială a bazei de date ... 60
18. Observații privind setările și mediile de lucru ... 60
19. Ce a adus Spring Data în mod concret ... 61
20. Întrebări recapitulative ... 61
21. Exerciții propuse ... 61
Laborator 3: Elevarea aplicației REST – DTO-uri, validări și tratare centralizată a erorilor ... 63
1. Ce se urmărește în această etapă ... 63
2. De ce apar DTO-uri separate pentru request și response ... 63
3. Structura proiectului în noua variantă ... 64
4. Entitatea rămâne modelul persistent ... 64
5. DTO-uri pentru request ... 66
6. DTO pentru response ... 68
7. Bean Validation ... 72
8. Mapper-ul dintre entitate și DTO-uri ... 77
9. Repository-ul rămâne simplu ... 81
10. Excepție dedicată pentru resursa inexistentă ... 81
11. DTO-uri pentru răspunsurile de eroare ... 82
12. Service-ul devine mai expresiv ... 84
13. Controller-ul devine mai zvelt (thin) ... 86
14. Tratarea centralizată a erorilor ... 87
15. Exemplu de răspuns pentru produs inexistent ... 89
16. Exemplu de răspuns pentru eroare de validare ... 90
17. Configurații și observații utile ... 90
18. Documentarea API-ului cu OpenAPI și Swagger UI ... 91
19. Ce s-a schimbat conceptual în această elevare ... 94
20. Avantajele noii variante ... 94
21. Codul complet, în ordinea logică a rolurilor ... 95
22. Utilizarea Lombok în aplicații Spring Boot ... 107
23. Întrebări recapitulative ... 120
24. Exerciții propuse ... 121
Laborator 4: De la funcționare la monitorizare – logging și metrici ... 122
1. Logging în aplicații Spring Boot ... 122
2. Integrarea logging-ului în aplicația REST ... 130
3. Colectarea și analiza logurilor cu Elastic Stack (ELK) folosind Docker ... 142
4. Metrici în aplicații Spring Boot ... 150
5. Întrebări recapitulative ... 164
6. Exercițiu – Analiza execuției aplicației REST prin loguri și metrici (Grafana) ... 164
Laborator 5: Spring Security – autentificare și autorizare într-un API REST ... 168
1. Obiectivul laboratorului ... 168
2. De ce securitate într-un API REST ... 168
3. Ce este Spring Security și cum funcționează ... 169
4. CSRF – Cross-Site Request Forgery ... 187
5. Managementul sesiunilor ... 188
6. Autentificare cu JWT (JSON Web Tokens) ... 189
7. Varianta alternativă: OAuth2 Resource Server cu JWT ... 197

8. CORS – Cross-Origin Resource Sharing ... 199
9. Tratarea erorilor de securitate ... 200
10. Plasarea regulilor de autorizare ... 202
11. Aplicarea regulilor pe aplicația restlab – exemplu consolidat ... 202
12. Testarea endpoint-urilor securizate ... 205
13. Probleme frecvente și modul de rezolvare ... 206
14. Întrebări recapitulative ... 207
15. Concluzii ... 208
16. Exerciții propuse ... 208
Laborator 6: Testarea automată a aplicațiilor REST cu Spring Boot ... 209
1. Obiectivul laboratorului ... 209
2. De ce avem nevoie de testare automată ... 209
3. Tipurile fundamentale de testare automată ... 210
4. Piramida testării ... 215
5. Concepte și terminologie consacrată ... 215
6. Test Doubles – taxonomia lui Meszaros ... 217
7. Mock vs Stub – distincția esențială ... 219
8. Tehnologii pentru testare în ecosistemul Spring Boot ... 220
9. Pregătirea proiectului pentru testare ... 229
10. Implementarea testelor ... 234
11. Code coverage cu JaCoCo ... 261
12. Bune practici și anti-patterns ... 263
13. Întrebări recapitulative ... 265
14. Exerciții propuse ... 267
Laborator 7: Introducere în React. Componente, JSX, props și state ... 269
1. Obiectivul laboratorului ... 269
2. De ce React ... 269
3. Instrumente necesare și instalarea lor ... 271
4. Pornirea proiectului cu Vite ... 274
5. Anatomia unei componente React ... 279
6. JSX în detaliu ... 280
7. Props – datele primite din exterior ... 282
8. Tratarea evenimentelor ... 288
9. Compoziția componentelor ... 289
10. Componente de prezentare vs componente cu logică ... 291
11. Aplicația laboratorului: catalogul de produse cu date hardcodate ... 292
12. Concluzii ... 296
13. Exerciții propuse ... 297
Laborator 8: Conectarea la API-ul REST. useEffect, axios și afișarea listei de produse ... 298
1. Obiectivul laboratorului ... 298
2. Modelul comunicării într-o aplicație SPA ... 299
3. Same-Origin Policy și CORS ... 299
4. Configurarea CORS în Spring Boot ... 301
5. Instrumentul de comunicare: axios ... 304
6. Hook-ul useEffect – efecte secundare ... 305
7. Prima componentă care încarcă date ... 308

8. Greșeli uzuale și cum se evită ... 311
9. Separarea pe straturi: modulul productApi ... 312
10. Variabile de mediu pentru baseURL ... 315
11. Aplicația finală a laboratorului ... 315
12. Exerciții propuse ... 317
Laborator 9: Operații CRUD complete. Formulare controlate și tratarea erorilor de validare ... 319
1. Obiectivul laboratorului ... 319
2. Formulare controlate în React ... 319
3. Formularul de creare a unui produs ... 322
4. Validarea pe două niveluri ... 327
5. Integrarea formularului cu lista de produse ... 329
6. Formularul de editare ... 330
7. Ștergerea unui produs ... 333
8. Feedback prin notificări (toast) ... 334
9. Aplicația finală a laboratorului ... 336
10. Exerciții propuse ... 337
Laborator 10: Organizarea aplicației. Routing, custom hooks și structura de proiect ... 339
1. Obiectivul laboratorului ... 339
2. Routing într-un SPA ... 340
3. Instalarea și configurarea React Router ... 340
4. Componenta Layout ... 342
5. Hook-urile useParams și useNavigate ... 344
6. Pagina 404 ... 346
7. Custom hooks – extracția logicii de date ... 347
8. Structura proiectului ... 350
9. Build de producție ... 352
10. Direcții pentru aprofundare ... 354
11. Exerciții propuse ... 355
12. Bibliografie recomandată ... 356

Laborator 1: Construirea unui API REST cu Spring Boot

1. Spring Framework, Spring Boot și Spring Initializr

1.1. Ce este Spring Framework?

Spring Framework este un framework modular pentru dezvoltarea aplicațiilor Java. Rolul său de bază este să ofere infrastructura prin care componentele aplicației pot fi create, configurate și conectate între ele într-un mod controlat. Spring nu este un server web și nici un generator de aplicații; el furnizează un *container* și un set de mecanisme pe baza cărora este organizată și executată aplicația. [2, 21]

Obiectele gestionate de containerul Spring sunt numite, în mod uzual, *bean*-uri. În loc ca fiecare clasă să decidă singură cum își construiește toate dependențele, aplicația descrie componentele și relațiile dintre ele, iar containerul Spring se ocupă de instanțiere, configurare, conectare și ciclul lor de viață.

Spring este modular. Nucleul său oferă containerul de componente și mecanismele de configurare, iar modulele framework-ului adaugă facilități pentru dezvoltare web prin *Spring MVC*, validare, acces la date și tranzacții, programare orientată pe aspecte și alte funcționalități de infrastructură. În jurul Spring Framework există un ecosistem mai larg de proiecte, precum *Spring Data* și *Spring Security*.

1.2. Inversiunea controlului și injecția dependențelor

Ideea centrală a Spring este **Inversion of Control** (IoC - inversiunea controlului). Într-o abordare tradițională, codul aplicației controlează direct construirea obiectelor de care are nevoie. Într-o aplicație bazată pe Spring, această responsabilitate este inversată: containerul creează obiectele și le furnizează acolo unde sunt necesare. [2, 21]

Mecanismul folosit cel mai frecvent pentru realizarea acestei idei este **Dependency Injection** (DI - injecția dependențelor). De exemplu, dacă un controller are nevoie de un service, controller-ul nu trebuie să creeze explicit acel service cu operatorul `new`. El declară dependența, iar Spring îi furnizează o instanță corespunzătoare, de regulă prin constructor.

Această separare reduce cuplarea dintre clase: componentele depind de contractele și colaboratorii de care au nevoie, nu de modul concret în care aceștia sunt creați. Ca efect, componentele pot fi înlocuite sau configurate mai ușor, iar testarea devine mai simplă deoarece dependențele pot fi substituite cu implementări de test.

1.3. De la Spring Framework la Spring Boot

Spring Framework oferă mecanismele fundamentale pentru organizarea aplicației, dar configurarea unui proiect complet presupune în continuare alegerea dependențelor potrivite, configurarea componentelor și pregătirea mediului de execuție. Pentru a reduce acest efort inițial și pentru a standardiza cazurile uzuale, ecosistemul Spring oferă **Spring Boot**. [1, 23]

Spring Boot este construit peste Spring Framework. El folosește mecanismele Spring și adaugă un set de convenții și mecanisme de configurare automată care permit pornirea rapidă a unei aplicații. Containerul Spring, bean-urile, injecția dependențelor și modulele framework-ului rămân la baza aplicației; Spring Boot simplifică în principal configurarea și asamblarea lor.

Într-un proiect web clasic bazat pe Spring, programatorul ar trebui să aleagă și să configureze explicit numeroase biblioteci, serverul web, mecanismele de serializare și o parte din infrastructura aplicației. Spring Boot simplifică această etapă prin câteva idei centrale:

- *starter-e* – dependențe agregate care aduc un set coerent de biblioteci pentru un anumit tip de aplicație; de exemplu, spring-boot-starter-web pentru aplicații web și REST;
- *auto-configuration* – configurarea automată a componentelor uzuale în funcție de bibliotecile existente în proiect și de proprietățile definite de aplicație;
- *server web embedded* – aplicația poate fi pornită direct ca program Java, fără instalarea separată a unui server de aplicații; în configurația web uzuală, serverul HTTP este inclus ca dependență;
- *configurare externalizată* – setările pot fi păstrate în `application.properties` sau `application.yml` și pot fi suprascrise pentru medii diferite;
- *convenții implicite rezonabile* – aplicația pornește cu o configurație funcțională, iar programatorul modifică numai elementele care diferă de cazul uzual.

Punctul de intrare într-o aplicație Spring Boot este, de regulă, o clasă marcată cu `@SpringBootApplication`. Această adnotare combină trei roluri importante: declară o sursă de configurație, activează mecanismul de auto-configurare și pornește scanarea componentelor Spring din pachetul aplicației și subpachetele sale. Metoda `SpringApplication.run(...)` construiește contextul Spring, creează bean-urile necesare și, pentru o aplicație web, pornește serverul embedded. (Figura 1)

1.4. Crearea proiectului cu Spring Initializr

Spring Initializr este generatorul oficial de proiecte Spring. Interfața disponibilă la `https://start.spring.io/` permite selectarea caracteristicilor de bază, după care generează o arhivă care conține structura proiectului și fișierele de build necesare. Initializr nu generează aplicația propriu-zisă; el generează un punct de plecare coerent și reproductibil. [1]

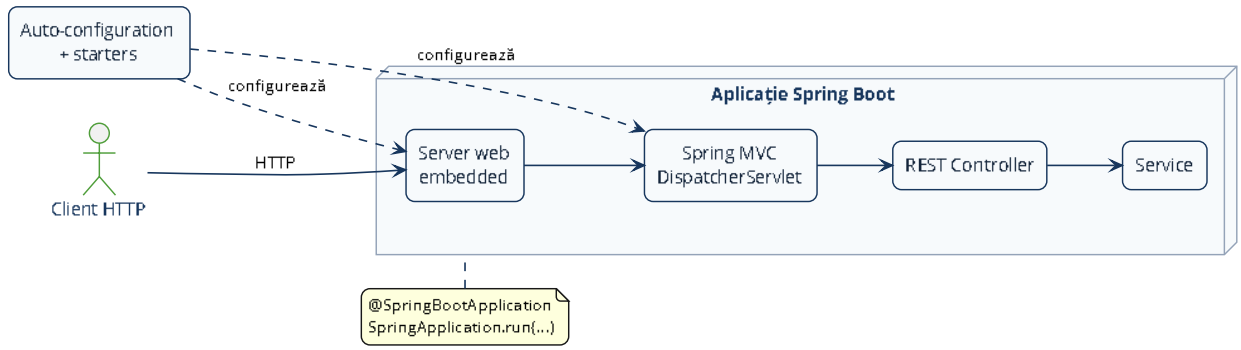


Figura 1 - Rolul Spring Boot într-o aplicație web REST

Pentru aplicația utilizată în acest laborator se poate folosi următoarea configurație:

1. se accesează `https://start.spring.io/`;
2. la *Project* se alege `Maven`;
3. la *Language* se alege `Java`;
4. se selectează o versiune stabilă de Spring Boot compatibilă cu versiunea Java utilizată în laborator;
5. la *Project Metadata* se pot utiliza, de exemplu, Group = `ro.unitbv` și Artifact = `restlab`;
6. la *Packaging* se alege `Jar`;
7. se selectează versiunea Java folosită de proiect;
8. prin *Add Dependencies* se adaugă `Spring Web`;
9. se apasă *Generate*, se dezarhivează proiectul și se importă în IDE ca proiect Maven.

Arhiva generată conține, între altele, fișierul `pom.xml`, Maven Wrapper și clasa principală a aplicației. Proiectul poate fi pornit direct din IDE sau, din directorul proiectului, cu Maven Wrapper. Pe Windows se poate utiliza `mvnw.cmd spring-boot:run`, iar pe Linux/macOS `./mvnw spring-boot:run`.

Documentația oficială Spring Boot și Spring Initializr rămâne sursa de referință pentru opțiunile și versiunile disponibile; bibliografia de la finalul îndrumarului include documentația Spring relevantă.

2. Obiectivul laboratorului

În acest laborator se urmărește înțelegerea modului în care se construiește, pas cu pas, un API REST simplu cu **Spring Boot**. Accentul cade pe lucrul cu **REST controllers**, pe maparea cererilor HTTP către metode Java și pe schimbul de date în format JSON[1].

Un **API REST** este o interfață prin care o aplicație oferă altor aplicații acces la resurse și operații prin protocolul HTTP, folosind URL-uri clare și metode standard precum `GET`, `POST`, `PUT`, `DELETE`. [34, 36]

[1] JSON (*JavaScript Object Notation*) este un format text simplu și ușor de citit, folosit pentru reprezentarea și schimbul de date structurate între aplicații.

Mai concret, într-un API REST:

- o **resursă** este ceva asupra căruia se poate lucra, de exemplu: produse, studenți, comenzi;
- fiecare resursă are de obicei un URL, de exemplu:
  - `/products`
  - `/products/10`
- operațiile se exprimă prin metode HTTP:
  - `GET` – citește date;
  - `POST` – creează;
  - `PUT` – actualizează;
  - `DELETE` – șterge.

Exemplu simplu:

- `GET /products` → returnează lista produselor
- `GET /products/5` → returnează produsul cu id 5
- `POST /products` → adaugă un produs nou
- `PUT /products/5` → modifică produsul 5
- `DELETE /products/`5 → șterge produsul 5

Se numește **REST** de la **Representational State Transfer** și descrie un stil arhitectural în care serverul expune resurse într-un mod standardizat, de regulă schimbând date în format JSON. [36]

Exemplele sunt construite gradual. La început se va folosi o variantă foarte simplă, în care toată logica este plasată direct în controller. Această alegere este intenționată: pentru primele exerciții este mai important ca mecanismul să fie clar decât ca arhitectura să fie perfectă. După ce funcționalitatea devine suficient de mare, vor fi observate limitele acestei abordări, iar aplicația va fi reorganizată în mod natural pe straturi.

3. Aplicația care va fi realizată

În cadrul acestui laborator se va construi un API REST simplu pentru gestionarea unor produse. Aplicația va permite efectuarea operațiilor de bază asupra unei colecții de produse, astfel încât să poată fi ilustrate, într-un context concret, principalele mecanisme de lucru cu `@RestController` în Spring Boot.

Fiecare produs va avea câteva atribute elementare, precum:

- un identificator;
- un nume;
- un preț;
- o cantitate în stoc.

Asupra acestor produse, aplicația va oferi operațiile uzuale:

- afișarea tuturor produselor;
- afișarea unui produs după identificator;
- adăugarea unui produs nou;
- modificarea unui produs existent;
- ștergerea unui produs.

Scopul nu este doar obținerea unei aplicații funcționale, ci și înțelegerea modului în care cererile HTTP sunt mapate către metode Java, cum sunt primite datele de la client și cum sunt returnate răspunsurile în format JSON.

4. Pornirea proiectului

Proiectul se generează cu Spring Initializr, folosind pașii prezentați în introducerea acestui capitol. Pentru laborator sunt suficiente Maven, Java și dependența Spring Web. O configurație minimală poate folosi Group = `ro.unitbv`, Artifact = `restlab`, Packaging = `Jar` și versiunea Java stabilită pentru laborator.

În proiectul generat, dependența principală va fi adusă prin starter-ul:

```
spring-boot-starter-web
```

Initializr generează și clasa principală, de forma:

RestLabApplication.java — src/main/java/ro/unitbv/restlab/RestLabApplication.java

```
package ro.unitbv.restlab;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestLabApplication {
    public static void main(String[] args) {
        SpringApplication.run(RestLabApplication.class, args);
    }
}
```

După importul proiectului în IDE, o structură minimă poate fi:

```
src/main/java/ro/unitbv/restlab
│
├── RestLabApplication.java
└── controller
    └── ProductController.java
```

Fișierul `pom.xml` este administrat de Maven și descrie dependențele proiectului. La această etapă nu este necesară configurarea unei baze de date; aplicația va porni doar cu infrastructura web. După pornire, serverul embedded ascultă în mod implicit pe portul 8080, dacă această valoare nu este modificată prin configurație.

Mai târziu, structura va fi extinsă pe măsură ce sunt introduse modelul, service-ul, repository-ul, persistența și mecanismele de securitate.

5. Punctul de plecare: un controller foarte simplu

Pentru început, este util să vedem cea mai simplă formă de controller. Nu ne grăbim cu model complex, cu servicii sau cu separări pe straturi. Ideea este să vedem mai întâi traseul unei cereri.

5.1. Primul controller

```
package ro.unitbv.restlab.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ProductController {

    @GetMapping("/hello")
    public String hello() {
        return "Salut din Spring Boot!";
    }
}
```

Dacă aplicația este pornită și se accesează în browser:

```
http://localhost:8080/hello
```

se va obține răspunsul:

```
Salut din Spring Boot!
```

5.2. Ce face @RestController

Adnotarea `@RestController` marchează clasa ca fiind un controller REST. În termeni practici, asta înseamnă două lucruri:

1. clasa este detectată de Spring și înregistrată ca o componentă a aplicației;
2. valorile returnate de metode sunt scrise direct în răspunsul HTTP.

Cu alte cuvinte, metoda `hello()` nu întoarce numele unei pagini, ci chiar conținutul răspunsului.

Într-un controller REST, răspunsurile sunt, de regulă:

- text simplu;
- obiecte Java transformate automat în JSON;
- liste de obiecte;
- răspunsuri HTTP controlate explicit prin `ResponseEntity`.

5.3. Ce face @GetMapping

Adnotarea `@GetMapping` spune că metoda va răspunde la cereri HTTP de tip **GET**. Ea face legătura dintre o cale URL și o metodă Java. [2]

În exemplul de mai sus:

```
@GetMapping("/hello")
```

înseamnă că orice cerere GET către `/hello` va fi preluată de metoda `hello()`.

6. Trecerea de la text la obiecte JSON

Un API REST util nu întoarce doar șiruri de caractere. În mod uzual, el transmite obiecte în format JSON. De aceea, următorul pas este să întoarcem un obiect Java.

6.1. Modelul Product

Vom adăuga o clasă simplă:

```
package ro.unitbv.restlab.controller;

public class Product {

    private Integer id;
    private String name;
    private double price;
    private int stock;

    public Product() { }

    public Product(Integer id, String name, double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

Pentru moment, chiar dacă această clasă ar merita un pachet separat, o păstrăm conceptual foarte aproape de controller ca să nu fragmentăm explicația prea devreme.

6.2. Un endpoint care întoarce un produs

```
package ro.unitbv.restlab.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class ProductController {

    @GetMapping("/product")
    public Product getSampleProduct() {
        return new Product(1, "Laptop", 3500.0, 10);
    }
}
```

La accesarea endpoint-ului:

```
GET /product
```

răspunsul va fi de forma:

```
{
  "id": 1,
  "name": "Laptop",
  "price": 3500.0,
  "stock": 10
}
```

6.3. Cum ajunge obiectul Java să devină JSON

În momentul în care metoda returnează un obiect Java, Spring folosește infrastructura de serializare pentru a transforma acel obiect în JSON. În mod normal, pentru Spring Boot acest lucru se face automat. [1, 2]

Practic:

- metoda returnează `Product`;
- Spring inspectează obiectul;
- proprietățile expuse prin getter-e sunt transformate în câmpuri JSON;
- rezultatul este trimis clientului în corpul răspunsului.

Acesta este unul dintre motivele pentru care controller-ele REST sunt atât de comode: programatorul lucrează cu obiecte Java obișnuite, iar framework-ul se ocupă de conversie.

7. Returnarea unei liste de obiecte

În aplicațiile reale, rareori se lucrează doar cu un singur obiect. De cele mai multe ori este nevoie de endpoint-uri care întorc colecții.

7.1. Un exemplu cu listă

```
package ro.unitbv.restlab.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.ArrayList;
import java.util.List;

@RestController
public class ProductController {

    @GetMapping("/products")
    public List<Product> getProducts() {
        List<Product> products = new ArrayList<>();
        products.add(new Product(1, "Laptop", 3500.0, 10));
        products.add(new Product(2, "Mouse", 120.0, 50));
        products.add(new Product(3, "Keyboard", 250.0, 30));
        return products;
    }
}
```

Răspunsul va fi:

```
[
  {
    "id": 1,
    "name": "Laptop",
    "price": 3500.0,
    "stock": 10
  },
```

```
    {
      "id": 2,
      "name": "Mouse",
      "price": 120.0,
      "stock": 50
    },
    {
      "id": 3,
      "name": "Keyboard",
      "price": 250.0,
      "stock": 30
    }
]
```

6.2. Observație importantă

În această versiune, lista este creată la fiecare apel. Evident, nu este o soluție potrivită pentru o aplicație reală, dar pentru înțelegerea mecanismului este suficientă: se vede foarte clar că un endpoint poate întoarce nu doar un obiect, ci și o colecție de obiecte.

8. Introducerea unei colecții persistente în memorie

Acum putem trece la un pas mai realist: să păstrăm datele într-o structură de date aflată la nivelul controller-ului. Astfel, endpoint-urile vor lucra cu aceeași colecție pe durata rulării aplicației.

Vom folosi un obiect de tip `ConcurrentHashMap<Integer, Product>`.

8.1. Controller cu „bază de date” în memorie

```
package ro.unitbv.restlab.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.ArrayList;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.List;

@RestController
public class ProductController {

    private final Map<Integer, Product> products =
                                        new ConcurrentHashMap<>();

    public ProductController() {
        products.put(1, new Product(1, "Laptop", 3500.0, 10));
        products.put(2, new Product(2, "Mouse", 120.0, 50));
        products.put(3, new Product(3, "Keyboard", 250.0, 30));
    }
```

```
    @GetMapping("/products")
    public List<Product> getProducts() {
        return new ArrayList<>(products.values());
    }
}
```

Acum endpoint-ul `/products` va returna conținutul colecției din memorie.

9. Lucrul cu identificatori din URL

Până aici am afișat toate produsele. Următorul pas natural este afișarea unui singur produs după id.

9.1. Endpoint-ul

```
@GetMapping("/products/{id}")
public Product getProductById(@PathVariable Integer id) {
    return products.get(id);
}
```

9.2. Ce face @PathVariable

Adnotarea `@PathVariable` extrage o valoare direct din calea URL.

În expresia:

```
@GetMapping("/products/{id}")
```

partea `{id}` este un segment variabil. Când clientul trimite o cerere către:

```
/products/2
```

Spring înțelege că valoarea 2 trebuie preluată și transmisă ca argument metodei:

```
@PathVariable Integer id
```

Asta înseamnă că URL-ul devine o sursă directă de parametri pentru metoda Java. [2]

9.3. Prima problemă: produs inexistent

Dacă produsul nu există, metoda `products.get(id)` va întoarce `null`. Spring poate transforma acest lucru într-un răspuns gol, dar aici apare deja o problemă de expresivitate: clientul nu primește suficient de clar un semnal corect că resursa nu există.

Într-un API REST este mai potrivit să controlăm explicit răspunsul HTTP.

10. Controlul răspunsului HTTP cu ResponseEntity

Aici apare natural nevoia de a întoarce nu doar un obiect, ci un răspuns complet: corp plus cod de stare.

10.1. Varianta îmbunătățită

```
import org.springframework.http.ResponseEntity;

@GetMapping("/products/{id}")
public ResponseEntity<Product> getProductById(@PathVariable Integer id) {
    Product product = products.get(id);

    if (product == null) {
        return ResponseEntity.notFound().build();
    }

    return ResponseEntity.ok(product);
}
```

10.2. Ce este ResponseEntity

`ResponseEntity` este un tip special care permite construirea explicită a răspunsului HTTP. Cu ajutorul lui se poate controla:

- codul de stare;
- corpul răspunsului;
- antetele HTTP.

În exemplul de mai sus:

- `ResponseEntity.ok(product)` înseamnă răspuns cu cod `200 OK` și corpul conținând produsul;
- `ResponseEntity.notFound().build()` înseamnă răspuns cu cod `404 Not Found` și fără corp.

Acesta este un moment important în înțelegerea controller-elor REST: nu totul înseamnă doar „returnează un obiect”; uneori este esențial să fie controlat semantic răspunsul HTTP. [2]

11. Primirea datelor de la client: @RequestBody

Până acum au fost expuse date. Următorul pas natural este să se primească date de la client pentru a crea produse noi.

11.1. Endpoint pentru adăugare

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;

@PostMapping("/products")
public Product createProduct(@RequestBody Product product) {
    products.put(product.getId(), product);
    return product;
}
```

11.2. Ce face @RequestBody

`@RequestBody` spune lui Spring că metoda așteaptă să primească date în corpul cererii HTTP și că acele date trebuie convertite într-un obiect Java. [2]

De exemplu, dacă clientul trimite:

```
POST /products
Content-Type: application/json
```

cu body:

```
{
  "id": 4,
  "name": "Monitor",
  "price": 999.99,
  "stock": 12
}
```

Spring va citi JSON-ul și va construi automat un obiect Product, pe care îl va transmite metodei ca argument.

Acest mecanism se numește **deserializare**: din JSON către obiect Java.

11.3. Construirea completă a exemplului

Merită observată evoluția:

1. clientul trimite un JSON;
2. Spring îl transformă într-un obiect Product;
3. metoda Java primește acel obiect;
4. controller-ul îl adaugă în colecție;
5. obiectul returnat de metodă este din nou transformat în JSON;
6. clientul primește răspunsul.

Astfel, fluxul complet este:

JSON → obiect Java → procesare → obiect Java → JSON

Aceasta este una dintre ideile centrale ale dezvoltării API-urilor cu Spring.

12. Alegerea codului de stare potrivit la creare

Dacă produsul a fost creat, codul de stare potrivit nu este neapărat `200 OK`. În mod uzual, pentru creare se folosește `201 Created`.

12.1. Variantă mai corectă

```
import org.springframework.http.HttpStatus;

@PostMapping("/products")
public ResponseEntity<Product> createProduct(@RequestBody Product product)
```

```
{
    products.put(product.getId(), product);
    return ResponseEntity.status(HttpStatus.CREATED).body(product);
}
```

12.2. Ce face HttpStatus.CREATED

`HttpStatus.CREATED` corespunde codului `HTTP 201`. El transmite mai clar semnificația operației: nu doar că cererea a fost procesată cu succes, ci că o resursă nouă a fost creată. [18, 35]

Într-un API bine construit, alegerea corectă a codurilor de stare este foarte importantă. Clientul nu interpretează doar corpul răspunsului, ci și codul HTTP.

13. Actualizarea unei resurse cu @PutMapping

După creare, următoarea operație naturală este actualizarea.

13.1. Endpoint-ul

```
import org.springframework.web.bind.annotation.PutMapping;

@PutMapping("/products/{id}")
public ResponseEntity<Product> updateProduct(@PathVariable Integer id,
                                             @RequestBody Product product)
{
    if (!products.containsKey(id)) {
        return ResponseEntity.notFound().build();
    }

    product.setId(id);
    products.put(id, product);

    return ResponseEntity.ok(product);
}
```

13.2. Ce face @PutMapping

`@PutMapping` mapează cererile HTTP de tip PUT. În mod uzual, PUT este asociat cu actualizarea unei resurse existente. [18, 35]

În exemplul nostru, clientul alege resursa prin URL:

```
/products/2
```

și trimite noua stare a acelei resurse în corpul cererii.

13.3. De ce se suprascrie id-ul cu cel din URL

În exemplu apare instrucțiunea:

```
product.setId(id);
```

Aceasta este importantă deoarece identificatorul resursei este deja stabilit prin URL. Dacă în corpul cererii ar apărea un alt id, ar apărea ambiguitate. Prin urmare, endpoint-ul decide că sursa principală pentru identificator este calea URL.

Aceasta este o convenție foarte frecventă în API-uri REST.

14. Ștergerea unei resurse cu @DeleteMapping

Ultima operație este ștergerea.

14.1. Endpoint-ul

```
import org.springframework.web.bind.annotation.DeleteMapping;

@DeleteMapping("/products/{id}")
public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
    Product removed = products.remove(id);

    if (removed == null) {
        return ResponseEntity.notFound().build();
    }

    return ResponseEntity.noContent().build();
}
```

14.2. Ce face @DeleteMapping

`@DeleteMapping` mapează cererile HTTP de tip DELETE. Semnificația este directă: clientul cere eliminarea unei resurse. [18]

14.3. Ce înseamnă ResponseEntity<Void>

Aici metoda nu mai returnează un obiect, ci doar un răspuns fără corp. De aceea tipul este:

```
ResponseEntity<Void>
```

`Void` indică faptul că nu există corp de răspuns relevant.

Codul:

```
ResponseEntity.noContent().build();
```

produce răspunsul `204 No Content`, foarte potrivit pentru o ștergere reușită.

15. Varianta completă: tot codul în controller

În acest moment, se poate reuni totul într-o versiune completă.

```
package ro.unitbv.restlab.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
```

```
import org.springframework.web.bind.annotation.*;

import java.util.ArrayList;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final Map<Integer, Product> products =
                                            new ConcurrentHashMap<>();

    public ProductController() {
        products.put(1, new Product(1, "Laptop", 3500.0, 10));
        products.put(2, new Product(2, "Mouse", 120.0, 50));
        products.put(3, new Product(3, "Keyboard", 250.0, 30));
    }

    @GetMapping
    public List<Product> getAllProducts() {
        return new ArrayList<>(products.values());
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(
                                            @PathVariable Integer id) {
        Product product = products.get(id);
        if (product == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(product);
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(
                                        @RequestBody Product product) {
        products.put(product.getId(), product);
        return ResponseEntity.status(HttpStatus.CREATED).body(product);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(
                                            @PathVariable Integer id,
                                            @RequestBody Product product) {
        if (!products.containsKey(id)) {
            return ResponseEntity.notFound().build();
        }
        product.setId(id);
        products.put(id, product);

        return ResponseEntity.ok(product);
    }
```

```
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        Product removed = products.remove(id);

        if (removed == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.noContent().build();
    }
}
```

16. Ce face @RequestMapping

Apare o adnotare nouă:

```
@RequestMapping("/api/products")
```

Aceasta definește o cale de bază pentru toate metodele din controller. Astfel:

- `@GetMapping` înseamnă de fapt `GET /api/products`
- `@GetMapping("/{id}")` înseamnă `GET /api/products/{id}`
- `@PostMapping` înseamnă `POST /api/products`
- `@PutMapping("/{id}")` înseamnă `PUT /api/products/{id}`
- `@DeleteMapping("/{id}")` înseamnă `DELETE /api/products/{id}`

Avantajul este clar: nu mai repetăm aceeași bază la fiecare metodă.

`@RequestMapping` poate fi folosit atât la nivel de clasă, cât și la nivel de metodă. În practică, la nivel de clasă este foarte util pentru stabilirea „rădăcinii” resursei. [2]

17. Testarea pas cu pas

După ce aplicația pornește, endpoint-urile pot fi testate.

17.1. Afișarea tuturor produselor

```
GET /api/products
```

Răspuns:

```
[
  {
    "id": 1,
    "name": "Laptop",
    "price": 3500.0,
    "stock": 10
  },
  {
    "id": 2,
    "name": "Mouse",
```

```
    "price": 120.0,
    "stock": 50
  },
  {
    "id": 3,
    "name": "Keyboard",
    "price": 250.0,
    "stock": 30
  }
]
```

17.2. Afișarea unui produs

```
GET /api/products/2
```

Răspuns:

```
{
  "id": 2,
  "name": "Mouse",
  "price": 120.0,
  "stock": 50
}
```

17.3. Crearea unui produs

```
POST /api/products
Content-Type: application/json
```

Body:

```
{
  "id": 4,
  "name": "Monitor",
  "price": 999.99,
  "stock": 12
}
```

17.4. Actualizarea unui produs

```
PUT /api/products/4
Content-Type: application/json
```

Body:

```
{
  "id": 100,
  "name": "Monitor 27",
  "price": 1199.99,
  "stock": 8
}
```

Rezultatul va avea id-ul 4, pentru că el este luat din URL.

16.5. Ștergerea

```
DELETE /api/products/4
```

18. Primele limite ale abordării

Până aici aplicația este suficient de clară. Totuși, exact acum încep să se vadă și problemele.

Controller-ul face prea multe lucruri:

- gestionează cererile HTTP;
- decide răspunsurile HTTP;
- păstrează datele în memorie;
- conține logica de căutare, creare, actualizare și ștergere.

La câteva metode, acest lucru pare acceptabil. Dar dacă aplicația crește, controller-ul devine rapid prea încărcat.

În acest moment apare natural întrebarea: nu ar fi mai bine ca partea de lucru cu datele să fie mutată într-o componentă separată?

19. Separarea pe straturi

Se vor introduce acum trei roluri distincte (Figura 2):

- **model** – obiectele aplicației;
- **service** – logica de business și manipularea datelor;
- **controller** – interfața HTTP.



Figura 2 - Separarea responsabilităților

Abia acum această separare are sens, pentru că există deja suficientă funcționalitate încât avantajele să fie vizibile. [38, 39]

19.1. Modelul

Fișier: `model/Product.java`

```
package ro.unitbv.restlab.model;

public class Product {

    private Integer id;
    private String name;
    private double price;
    private int stock;
```

```
    public Product() { }

    public Product(Integer id, String name, double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

19.2. Service-ul

Fișier: `service/ProductService.java`

```
package ro.unitbv.restlab.service;

import org.springframework.stereotype.Service;
import ro.unitbv.restlab.model.Product;

import java.util.ArrayList;
import java.util.List;
```

```
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@Service
public class ProductService {

    private final Map<Integer, Product> products = new
ConcurrentHashMap<>();

    public ProductService() {
        products.put(1, new Product(1, "Laptop", 3500.0, 10));
        products.put(2, new Product(2, "Mouse", 120.0, 50));
        products.put(3, new Product(3, "Keyboard", 250.0, 30));
    }

    public List<Product> findAll() {
        return new ArrayList<>(products.values());
    }

    public Product findById(Integer id) {
        return products.get(id);
    }

    public Product save(Product product) {
        products.put(product.getId(), product);
        return product;
    }

    public Product update(Integer id, Product product) {
        if (!products.containsKey(id)) {
            return null;
        }

        product.setId(id);
        products.put(id, product);
        return product;
    }

    public boolean deleteById(Integer id) {
        return products.remove(id) != null;
    }
}
```

19.2.1. Ce face @Service

Adnotarea `@Service` marchează clasa drept componentă de tip service. Din punctul de vedere al containerului Spring, este o clasă gestionată automat, care poate fi injectată în alte componente.

Din punct de vedere arhitectural, un service este locul în care se păstrează logica aplicației. Controller-ul nu mai trebuie să știe cum sunt stocate datele sau cum se realizează operațiile asupra lor. El doar cere service-ului să facă aceste lucruri. [2]

19.3. Controller-ul simplificat

Fișier: `controller/ProductController.java`

```
package ro.unitbv.restlab.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.service.ProductService;

import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(
                                                @PathVariable Integer id) {
        Product product = productService.findById(id);

        if (product == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(product);
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(
                                        @RequestBody Product product) {
        Product savedProduct = productService.save(product);
        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(savedProduct);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(
                                        @PathVariable Integer id,
                                        @RequestBody Product product) {
        Product updatedProduct = productService.update(id, product);
```

```
            if (updatedProduct == null) {
                return ResponseEntity.notFound().build();
            }
            return ResponseEntity.ok(updatedProduct);
        }

        @DeleteMapping("/{id}")
        public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
            boolean deleted = productService.deleteById(id);

            if (!deleted) {
                return ResponseEntity.notFound().build();
            }
            return ResponseEntity.noContent().build();
        }
    }
```

20. Ce se câștigă prin separare

Acum se vede clar evoluția:

- controller-ul se ocupă de HTTP;
- service-ul se ocupă de operațiile asupra produselor;
- modelul descrie datele.

Avantajele devin vizibile imediat:

- codul este mai clar;
- clasele sunt mai scurte;
- responsabilitățile sunt mai bine separate;
- modificările devin mai ușor de făcut;
- testarea devine mai simplă.

21. O nouă problemă: obiectul expus la exterior coincide cu obiectul intern

Până acum, clasa `Product` este folosită peste tot:

- în memorie;
- în service;
- în controller;
- în request-uri;
- în response-uri.

La început pare convenabil. Dar, pe măsură ce aplicația crește, apare o întrebare foarte importantă: este oare bine ca obiectul intern al aplicației să fie exact același cu obiectul expus prin API?

Aici apare, în mod natural, nevoia de **DTO-uri**.

22. Obiecte DTO

DTO vine de la **Data Transfer Object**. Un DTO este un obiect folosit pentru transportul datelor între client și server. [38]

Ideea principală este simplă: obiectele cu care lucrează intern aplicația nu trebuie neapărat să fie identice cu obiectele trimise pe rețea.

22.1. Un prim exemplu simplu

Să presupunem că nu vrem ca la creare clientul să furnizeze `id`, ci doar:

- nume;
- preț;
- stoc.

Atunci putem crea un DTO special pentru cererea de creare.

Fișier: `dto/CreateProductRequest.java`

```
package ro.unitbv.restlab.dto;

public class CreateProductRequest {

    private String name;
    private double price;
    private int stock;

    public CreateProductRequest() {
    }

    public CreateProductRequest(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
```

```
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

22.2. De ce este util acest pas

Acum request-ul pentru creare nu mai este obligat să semene perfect cu modelul intern. Controller-ul începe să exprime mai bine intenția operației.

Clientul trimite doar datele necesare pentru creare, nu o copie completă a modelului intern.

22.3. Adaptarea service-ului

Adăugăm un mecanism simplu de generare a identificatorului.

```
package ro.unitbv.restlab.service;

import org.springframework.stereotype.Service;
import ro.unitbv.restlab.dto.CreateProductRequest;
import ro.unitbv.restlab.model.Product;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.atomic.AtomicInteger;

@Service
public class ProductService {

    private final Map<Integer, Product> products =
                                            new ConcurrentHashMap<>();
    private final AtomicInteger nextId = new AtomicInteger(4);

    public ProductService() {
        products.put(1, new Product(1, "Laptop", 3500.0, 10));
        products.put(2, new Product(2, "Mouse", 120.0, 50));
        products.put(3, new Product(3, "Keyboard", 250.0, 30));
    }

    public List<Product> findAll() {
        return new ArrayList<>(products.values());
    }

    public Product findById(Integer id) {
        return products.get(id);
    }

    public Product create(CreateProductRequest request) {
        Integer id = nextId.getAndIncrement();
        Product product = new Product(id, request.getName(),
```

```
                                        request.getPrice(),
                                        request.getStock());
        products.put(id, product);
        return product;
    }

    public Product update(Integer id, Product product) {
        if (!products.containsKey(id)) {
            return null;
        }

        product.setId(id);
        products.put(id, product);
        return product;
    }

    public boolean deleteById(Integer id) {
        return products.remove(id) != null;
    }
}
```

22.4. Adaptarea controller-ului

```
package ro.unitbv.restlab.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.CreateProductRequest;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.service.ProductService;

import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(
                                            @PathVariable Integer id) {
        Product product = productService.findById(id);
```

```
        if (product == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(product);
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(
                            @RequestBody CreateProductRequest request) {
        Product createdProduct = productService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(createdProduct);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(
                                        @PathVariable Integer id,
                                        @RequestBody Product product) {
        Product updatedProduct = productService.update(id, product);

        if (updatedProduct == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(updatedProduct);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        boolean deleted = productService.deleteById(id);

        if (!deleted) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.noContent().build();
    }
}
```

22.5. De la DTO-uri la modelarea mai atentă a domeniului

În acest punct apare foarte natural o idee mai profundă: poate că obiectele din sistem nu trebuie gândite doar ca simple containere de date. Poate că ele exprimă și concepte din domeniul problemei. Aici începe să fie util un prim contact cu câteva idei din **DDD – Domain-Driven Design**.

23. Primele idei de DDD

23.1. Entitatea

`Product` poate fi privit ca o **entitate**. O entitate este un obiect care are identitate proprie. În cazul nostru, identitatea este dată de `id`. [37]

Două produse pot avea același nume, același preț și același stoc, dar dacă au identificatori diferiți, ele rămân obiecte distincte.

Această idee devine importantă mai ales când aplicația începe să crească și obiectele nu mai sunt tratate doar ca date brute.

23.2. Separarea dintre modelul domeniului și contractul API

DTO-urile au arătat deja că obiectele trimise prin API nu trebuie să coincidă perfect cu entitățile domeniului. [37, 39]

Aceasta este o idee foarte sănătoasă din punct de vedere arhitectural:

- modelul domeniului exprimă logica internă și regulile aplicației;
- DTO-urile exprimă contractul de comunicare cu exteriorul.

Când aceste două niveluri sunt separate, aplicația devine mai flexibilă.

23.3. Reguli de business și comportament

La început, `Product` este doar o clasă cu câmpuri și getter-e/setter-e. Dar într-un model mai matur ar putea avea și comportament:

- modificarea stocului;
- verificarea validității prețului;
- reguli privind denumirea;
- restricții de actualizare.

În acel moment nu mai vorbim doar despre „date”, ci despre un obiect de domeniu care știe ceva despre propria sa stare.

Aceasta este una dintre direcțiile naturale în care duce DDD.

24. Întrebări recapitulative

1. Ce rol are adnotarea `@RestController`?
2. Ce diferență există între a returna direct un obiect și a returna `ResponseEntity<T>`?
3. Ce face `@GetMapping`?
4. Ce face `@PostMapping`?
5. Cum funcționează `@PathVariable`?
6. Cum funcționează `@RequestBody`?
7. Ce rol are `@RequestMapping` la nivel de clasă?
8. De ce este utilă separarea pe straturi?
9. De ce pot fi utile DTO-urile?
10. În ce sens poate fi `Product` privit ca o entitate?

25. Exerciții propuse

25.1. Exercițiul 1

Adăugați un endpoint:

```
GET /api/products/count
```

care întoarce numărul total de produse.

25.2. Exercițiul 2

Adăugați un endpoint:

```
GET /api/products/search/{name}
```

care caută produse după nume.

25.3. Exercițiul 3

Creați un DTO separat pentru răspuns, de exemplu `ProductResponse`, și adaptați endpoint-urile astfel încât controller-ul să nu mai trimită direct obiecte `Product`.

25.4. Exercițiul 4

Adăugați în service o regulă simplă: prețul trebuie să fie strict pozitiv, iar stocul nu poate fi negativ. Decideți cum va transmite controller-ul răspunsul HTTP în cazul datelor invalide.

25.5. Exercițiul 5

Gândiți o extindere a modelului astfel încât produsul să aibă și o categorie. Decideți dacă această categorie trebuie tratată doar ca text sau ca obiect separat.

Laborator 2: Persistența datelor cu Spring Data JPA și MySQL în aplicații REST Spring Boot

1. Obiectivul laboratorului

În laboratorul anterior a fost construit un API REST simplu pentru administrarea produselor, iar datele au fost păstrate în memorie, într-o colecție Java. Soluția a fost potrivită pentru înțelegerea controller-elor REST, a lucrului cu JSON și a operațiilor CRUD, însă aplicația avea o limitare evidentă: datele existau doar cât timp aplicația rula.

În cadrul acestui laborator, aceeași aplicație este extinsă astfel încât produsele să fie stocate într-o bază de date relațională. Pentru acest pas se vor introduce:

- *Spring Data JPA* pentru accesul la date;
- *MySQL* ca sistem de gestiune a bazei de date;
- configurarea conexiunii în *Spring Boot*;
- modelarea unei entități persistente;
- definirea unui repository;
- adaptarea service-ului și a controller-ului.

La final, aplicația va continua să ofere aceleași endpoint-uri REST, dar datele nu vor mai fi volatile, ci salvate în baza de date.

2. Ce este Spring Data și ce problemă rezolvă

În aplicațiile enterprise, stratul de persistență devine rapid o sursă majoră de cod repetitiv: deschiderea și închiderea contextului de persistență, operații CRUD[2] aproape identice, interogări simple scrise manual, mapări între rezultate și obiecte de domeniu, paginare, sortare și tratarea tranzacțiilor. **Spring Data** a fost creat pentru a standardiza aceste activități și pentru a oferi o abstracție unitară peste multiple tipuri de stocare: baze relaționale, document store, key-value store, grafuri și altele. [5, 38]

Nucleul conceptual se află în **Spring Data Commons**. Acesta definește abstracțiile de bază pentru repository-uri, paginare, sortare, Query by Example, evenimente de domeniu și infrastructura de proxy. Modulele concrete – de exemplu Spring Data JPA[3], Spring Data

[2] CRUD este un acronim care reprezintă cele patru operațiuni fundamentale de gestionare a datelor dintr-o aplicație: Create (Creare), Read (Citire), Update (Actualizare) și Delete (Ștergere).

[3] JPA (Java Persistence API) este un set de specificații standard care definește modul în care obiectele Java sunt mapate și salvate într-o bază de date relațională, eliminând necesitatea scrierii manuale a interogărilor SQL complexe.

MongoDB sau Spring Data JDBC – construiesc peste acest nucleu și traduc operațiile abstracte către mecanismele specifice fiecărui datastore.

În practică, cel mai frecvent caz este **Spring Data JPA**. Aici, repository-ul declarat de programator este doar o interfață; implementarea efectivă este creată de infrastructura **Spring Data** la pornirea aplicației, iar execuția reală a operațiilor se bazează pe **EntityManager**[4] și pe providerul JPA configurat, cel mai des **Hibernate**[5].

2.1. Unde se poziționează Spring Data în arhitectura aplicației

Într-o arhitectură clasică pe straturi, controller-ul primește cererea HTTP, serviciul aplică logica de business, iar repository-ul mediază accesul la modelul persistent. Spring Data nu ar trebui confundat cu stratul de servicii: repository-ul se ocupă de persistență și interogare, în timp ce serviciul orchestrează scenariile de utilizare, regulile de business, tranzacțiile cu granularitate mai mare și colaborarea între mai multe repository-uri. [38]

Din acest motiv, o practică sănătoasă este ca repository-urile să fie simple și centrate pe date, iar serviciile să conțină deciziile de business. În particular, nu este recomandat ca logica de validare complexă, agregările de business sau apelurile către sisteme externe să fie împinse în repository.

2.2. Cum funcționează intern infrastructura Spring Data

La pornirea aplicației, Spring scanează pachetele configurate pentru repository-uri. Pentru fiecare interfață care extinde unul dintre tipurile repository cunoscute, infrastructura creează un bean proxy[6]. Proxy-ul interceptează apelurile de metodă și decide cum trebuie rezolvată fiecare metodă. [5]

Rezolvarea se face, conceptual, în trei pași. Mai întâi se identifică semnătura metodei și tipul returnat. Apoi se stabilește strategia de căutare a interogării: metodă derivată din nume, interogare declarată prin adnotare sau named query[7]. În final, query-ul este executat prin infrastructura specifică modulului, în cazul de față Spring Data JPA și EntityManager.

Strategia implicită pentru rezolvarea interogărilor este `CREATE_IF_NOT_FOUND`: dacă există o interogare declarată, aceasta este folosită; altfel, Spring Data încearcă să derive query-ul din numele metodei. Această combinație este extrem de productivă pentru interogările simple, dar nu trebuie folosită mecanic atunci când expresivitatea numelui începe să degradeze lizibilitatea.

---

[4] EntityManager este o interfață din Java Persistence API (JPA) care servește ca un „manager de obiecte”, fiind responsabilă pentru gestionarea întregului ciclu de viață al entităților și pentru efectuarea operațiunilor CRUD asupra bazei de date.

[5] Hibernate este un framework de tip ORM (Object-Relational Mapping) care implementează specificația JPA, realizând maparea automată a claselor Java pe tabelele unei baze de date relaționale și gestionând execuția interogărilor SQL în locul programatorului.

[6] Proxy: Un obiect intermediar creat de Spring care "împachetează" bean-ul original pentru a-i adăuga funcționalități suplimentare (precum securitatea sau gestiunea tranzacțiilor) fără a-i modifica codul sursă.

[7] NamedQuery este o interogare statică definită prin adnotări la nivel de entitate, care este precompilată și validată de Hibernate la pornirea aplicației pentru a asigura o execuție mai rapidă și un cod mai curat.

2.3. Dependențe și pornirea unui proiect Spring Data JPA

Într-un proiect Spring Boot tipic, suportul pentru persistență relațională este activat prin starter-ul `spring-boot-starter-data-jpa`. Acesta aduce Spring Data JPA, Spring ORM, tranzacționalitatea declarativă și dependențele necesare integrării cu providerul JPA. Pentru dezvoltare și testare se adaugă de regulă și un driver JDBC, de exemplu H2, PostgreSQL sau MySQL. [1, 5]

Avantajul major al starter-ului este că versiunea componentelor este gestionată de Spring Boot. Programatorul specifică intenția funcțională - acces la date prin JPA - iar Boot rezolvă matricea de versiuni compatibile.

2.4. Configurarea prin application.properties și application.yml

Configurarea accesului la date aparține, în mod normal, fișierelor de configurare externe. În Spring Boot se pot folosi atât `application.properties`, cât și `application.yml`. Din perspectivă semantică, cele două formate exprimă aceleași chei; diferența este doar de sintaxă. Pentru coerență, este recomandat să se aleagă un singur format pentru întregul proiect. [1]

Pentru Spring Data JPA, proprietățile critice descriu sursa de date și comportamentul JPA/Hibernate: URL-ul JDBC, credențialele, dialectul, modul de creare a schemei, afișarea SQL-ului și parametrii adiționali transmiși providerului JPA.

Un exemplu minim de configurare pentru dezvoltare, în format `properties`, este prezentat mai jos.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/magazin
spring.datasource.username=app_user
spring.datasource.password=secret
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false
```

Aceeași configurare, în `YAML`, devine:

```
spring:
  datasource:
    url: jdbc:postgresql://localhost:5432/magazin
    username: app_user
    password: secret
    driver-class-name: org.postgresql.Driver

  jpa:
    hibernate:
      ddl-auto: validate
    show-sql: true
    open-in-view: false
```

```
    properties:
      hibernate:
        format_sql: true
```

2.5. Cum se corelează proprietățile cu profilurile

Profilurile Spring permit activarea selectivă a unei porțiuni de configurare pentru un anumit **mediu de execuție**. În practică, aceeași aplicație poate avea cerințe radical diferite în dezvoltare, testare și producție: baze de date distincte, niveluri diferite de logging, strategii diferite pentru schema bazei de date sau chiar bean-uri diferite. [1]

Mecanismul de bază este simplu: Spring Boot citește `application.properties` sau `application.yml`, apoi suprapune peste acesta fișierul specific profilului activ, de tipul `application-dev.properties`, `application-test.properties` sau `application-prod.properties`. Valorile din sursa cu prioritate mai mare le suprascriu pe cele anterioare.

Un model util este următorul: fișierul de bază conține valorile comune tuturor mediilor, iar fișierele pe profil conțin numai diferențele. În acest fel, configurația rămâne clară și se evită duplicarea inutilă.

Fișierul de bază:

```
# application.properties - valori comune
spring.application.name=demo-spring-data
spring.jpa.open-in-view=false
spring.jpa.properties.hibernate.format_sql=true

# activarea profilului
spring.profiles.active=dev
```

Configurație pentru profilul dev

```
# application-dev.properties
spring.datasource.url=jdbc:h2:mem:demo
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
```

Configurație pentru profilul prod:

```
# application-prod.properties
spring.datasource.url=${DB_URL}
spring.datasource.username=${DB_USERNAME}
spring.datasource.password=${DB_PASSWORD}
spring.jpa.hibernate.ddl-auto=validate
spring.jpa.show-sql=false
```

> **Regulă practică**
>
> Nu se activează de regulă profilul de producție direct în codul sursă. În producție, profilul este setat prin variabile de mediu, proprietăți de sistem sau argumente la pornire, pentru a separa aplicația de infrastructură.

2.5.1. Ordinea de suprascriere

În Spring Boot, fișierele de configurare nu sunt singura sursă de proprietăți. Valorile pot veni și din variabile de mediu, proprietăți JVM, argumente de linie de comandă sau locații externe de configurare. Principiul important este că sursele cu prioritate mai mare pot suprascrie valorile definite anterior.

De exemplu, dacă `application-prod.properties` definește `spring.datasource.url`, iar la pornire se transmite `--spring.datasource.url=...`, atunci valoarea din argumentul de linie de comandă are prioritate. Această regulă explică de ce aceeași aplicație poate fi redistribuită între medii fără recompilare.

2.5.2. Corelarea proprietăților cu bean-uri dependente de profil

Profilurile nu controlează doar fișierele `application-{profile}.properties`. Orice componentă, configurație sau clasă de proprietăți poate fi marcată cu `@Profile`. Astfel, se poate încărca un bean doar în mediul de dezvoltare sau doar în producție.

Această corelare este relevantă și pentru `@ConfigurationProperties`. Se poate avea o clasă de proprietăți comună tuturor mediilor sau, la nevoie, un bean diferit pe profil, cu prefix identic, dar cu procesare specializată.

De exemplu, activarea condiționată prin `@Profile` se poate face astfel:

```
@Configuration
@Profile("dev")
class DevelopmentDataInitializer {
    // bean-uri active doar în mediul de dezvoltare
}
```

2.6. Entități JPA și relația lor cu Spring Data

Spring Data JPA lucrează peste modelul JPA. De aceea, înainte de a discuta repository-urile, trebuie clarificat faptul că obiectele persistente sunt entități JPA, nu clase arbitrare. O entitate este o clasă adnotată cu `@Entity`, având un identificator stabil definit prin `@Id` și, frecvent, generat prin `@GeneratedValue`. [5, 6]

Spring Data nu inventează modelul de persistență, ci consumă metadatele JPA deja existente. Altfel spus, repository-ul operează asupra entităților și a identificatorilor lor.

O entitate JPA minimală poate fi:

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 120)
    private String name;

    @Column(nullable = false)
    private BigDecimal price;

    private boolean active;

    // constructori, getteri, setteri
}
```

Adnotările JPA frecvent întâlnite în acest context sunt:

Adnotare	***Nivel***	***Rol esențial***
`@Entity`	JPA	Marchează o clasă persistentă gestionată de providerul JPA.
`@Table`	JPA	Asociază explicit entitatea cu o tabelă și permite configurări suplimentare.
`@Id`	JPA	Definește cheia primară a entității.
`@GeneratedValue`	JPA	Precizează strategia de generare a identificatorului.
`@Column`	JPA	Controlează numele coloanei și constrângeri precum `nullable` sau `length`.
`@OneToMany / @ManyToOne`	JPA	Modelează relațiile între entități.

2.7. Ierarhia interfețelor Repository

Interfața de bază este `Repository<T, ID>`. Ea funcționează în principal ca interfață marker și stabilește tipul entității și tipul identificatorului. Pentru operații efective, se folosesc de regulă interfețe derivate.

`CrudRepository` oferă operațiile canonice create, read, update și delete. `ListCrudRepository` păstrează aceeași filosofie, dar returnează `List` în loc de `Iterable` pentru metodele care întorc mai multe entități, ceea ce este adesea mai comod în codul aplicației. [5]

`PagingAndSortingRepository` introduce paginarea și sortarea. `JpaRepository` extinde această familie și adaugă capabilități utile pentru JPA, precum `flush()`, `saveAndFlush()` sau operații batch. La acestea se pot adăuga interfețe suplimentare, cum ar fi `JpaSpecificationExecutor` sau `QueryByExampleExecutor`, pentru scenarii dinamice de interogare.

Interfață	*Ce oferă*	*Când este aleasă*
`Repository<T, ID>`	Tipuri de domeniu și ID; fără CRUD complet	Când se dorește expunerea selectivă a unor metode
`CrudRepository<T, ID>`	save, findById, existsById, findAll, delete...	Cazuri standard cu necesar CRUD
`ListCrudRepository<T, ID>`	CRUD, dar cu List pentru colecții	Când se preferă API mai ergonomic
`PagingAndSortingRepository <T, ID>`	findAll(Pageable), findAll(Sort)	Liste mari, UI paginat, API-uri REST
`JpaRepository<T, ID>`	Tot ce este mai sus plus flush și facilități JPA	Alegerea implicită în majoritatea proiectelor JPA
`JpaSpecificationExecutor<T>`	Interogări dinamice cu Specification	Filtrare complexă și compunere de predicate
`QueryByExampleExecutor<T>`	Interogare pornind de la un obiect-exemplu	Formulare de căutare simple și dinamice

Exemplu de Repository compus pentru un model de produs:

```
public interface ProductRepository
        extends JpaRepository<Product, Long>,
                JpaSpecificationExecutor<Product>,
                QueryByExampleExecutor<Product> {
}
```

2.8. Metodele de bază și semantica lor

Metoda `save(...)` nu înseamnă obligatoriu `INSERT`. În JPA, operația exactă depinde de starea entității și de identificator: poate însemna persist, merge sau actualizarea unei entități deja gestionate. Acesta este unul dintre motivele pentru care trebuie înțeles și contextul JPA, nu doar semnătura metodei repository. [6]

Metodele `findById(...)` și `getReferenceById(...)` nu sunt echivalente. Prima încarcă efectiv entitatea sau întoarce `Optional.empty()` dacă nu există. A doua poate întoarce o referință proxy și este utilă atunci când se dorește doar asocierea unei entități existente sau deferirea încărcării.

În cazul `delete(...)`, comportamentul final este dependent de contextul tranzacțional și de relațiile modelate. Cascadele și *orphan removal* țin de JPA și trebuie proiectate conștient; Spring Data doar delegă operația.

2.9. Derivarea interogărilor din numele metodei

Una dintre cele mai productive facilități Spring Data este query derivation. Programatorul declară o metodă în interfață, iar infrastructura construiește interogarea pe baza numelui. Prefixe uzuale sunt `findBy`, `readBy`, `getBy`, `existsBy`, `countBy` și `deleteBy`. [5]

Parserul descompune numele metodei în proprietăți și operatori. De exemplu, `findByNameAndActiveTrueOrderByPriceAsc` se traduce într-o interogare care filtrează după `name` și `active=true`, apoi sortează crescător după `price`. Mecanismul este extrem de util pentru predicate scurte și clare.

Totuși, query derivation are o limită de expresivitate. Dacă numele metodei devine excesiv de lung, greu de citit sau ascunde reguli complexe, este preferabilă trecerea la `@Query`, la `Specification` sau la o implementare custom.

Fragment în nume	*Semnificație uzuală*	*Exemplu*
`FindBy / ReadBy / GetBy`	Selectare entități	`findByEmail(String email)`
`And / Or`	Compunere de predicate	`findByLastNameAndActive(...)`
`Between`	Interval	`findByPriceBetween(min, max)`
`LessThan / GreaterThan`	Comparații numerice	`findByStockGreaterThan(int n)`
`Containing / StartingWith / EndingWith`	Căutări pe șiruri	`findByNameContaining(String x)`
`True / False`	Predicate booleene	`findByActiveTrue()`
`OrderBy...Asc/Desc`	Sortare statică	`findByActiveTrueOrderByPriceDesc()`
`Top / First`	Limitare statică	`findTop10ByOrderByCreatedAtDesc()`

Exemple de metode derivate:

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    List<Product> findByActiveTrue();
    Optional<Product> findByNameIgnoreCase(String name);
    List<Product> findTop5ByActiveTrueOrderByPriceDesc();
    Page<Product> findByNameContainingIgnoreCase(String fragment,
                                                 Pageable pageable);
    boolean existsByNameIgnoreCase(String name);
}
```

2.10. Interogări declarate cu @Query

Atunci când numele metodei nu mai poate exprima elegant logica de filtrare sau când este necesar controlul explicit asupra JPQL-ului[8] ori SQL-ului nativ, se folosește `@Query`. În Spring Data JPA, `@Query` poate descrie interogări JPQL sau, dacă se activează `nativeQuery`, interogări SQL native. [5]

`@Param` permite asocierea explicită a parametrilor metodei cu parametrii numiți din query. În practică, aceasta crește lizibilitatea și reduce fragilitatea codului la reordonarea parametrilor.

Pentru operații de update sau delete care nu întorc entități, se utilizează de regulă `@Modifying` împreună cu `@Query`. Fără `@Modifying`, infrastructura tratează metoda ca interogare de selecție.

Exemple de utilizare `@Query`, `@Param` și `@Modifying`:

[8] JPQL (Java Persistence Query Language) este limbajul de interogare definit de specificația JPA, utilizat pentru a formula interogări asupra entităților și relațiilor dintre acestea, independent de schema concretă a bazei de date.

```
public interface ProductRepository extends JpaRepository<Product, Long> {

    @Query("""
           select p
           from Product p
           where p.active = true
             and p.price >= :minPrice
           order by p.price desc
           """)
    List<Product> findActiveProductsWithMinimumPrice(
                            @Param("minPrice") BigDecimal minPrice);

    @Modifying
    @Transactional
    @Query("update Product p set p.active = false where p.price = 0")
    int deactivateFreeProducts();
}
```

2.11. Paginare, sortare și rezultate mari

Pentru colecții mari, returnarea întregului set de rezultate este ineficientă și, adesea, imposibilă din motive de memorie și latență. Spring Data Commons definește tipurile `Pageable`, `Page`, `Slice` și `Sort` pentru controlul acestor scenarii.

`Page` conține atât conținutul curent, cât și metadate complete despre numărul total de elemente și pagini. `Slice` este mai simplu: indică doar dacă mai există date următoare, evitând un count query costisitor în unele situații. `Sort` poate fi folosit separat sau împreună cu `Pageable`. [5]

Pentru fluxuri foarte mari de date, modulele Spring Data oferă și mecanisme de scrolling. În astfel de scenarii, cheia este alegerea unei ordonări stabile și a unui model de acces incremental, nu încărcarea brută a întregii tabele.

Exemple de paginare și sortare:

```
Pageable pageable = PageRequest.of(0, 20, Sort.by("price").descending());

Page<Product> page = productRepository
        .findByNameContainingIgnoreCase("laptop", pageable);
```

2.12. Proiecții și reducerea volumului de date transferat

Nu orice interogare trebuie să întoarcă entitatea completă. Când interfața utilizatorului afișează doar câteva câmpuri, proiecțiile reduc traficul, costul materializării și cuplarea dintre UI și modelul persistent.

Spring Data JPA suportă proiecții bazate pe interfețe și proiecții bazate pe clase. Primele sunt comode și concise; celelalte sunt utile când se dorește un DTO explicit, eventual imutabil. Alegerea trebuie să țină cont și de forma query-ului: pentru DTO-uri bazate pe clasă, JPQL folosește *constructor expressions*. [5]

Exemplu de proiecție bazată pe interfață:

```
public interface ProductSummary {
    String getName();
    BigDecimal getPrice();
}

public interface ProductRepository extends JpaRepository<Product, Long> {
    List<ProductSummary> findByActiveTrue();
}
```

2.13. Query by Example și Specification

Query by Example este util atunci când există formulare de căutare simple, iar criteriile sunt parțial opționale. Programatorul construiește un obiect-exemplu și un `ExampleMatcher`; infrastructura generează query-ul pornind de la câmpurile populate. [5]

Limitarea majoră a QBE este expresivitatea: scenariile cu grupări logice, predicate și combinații sofisticate nu se exprimă bine. Pentru acestea, în Spring Data JPA, *Specification* reprezintă soluția mai puternică.

În practică, QBE este excelent pentru filtre administrative simple; Specification devine preferabilă în API-uri de căutare avansată, unde filtrele sunt dinamice și combinabile.

Exemplu de *Query by Example*:

```
Product probe = new Product();
probe.setActive(true);
probe.setName("lap");

ExampleMatcher matcher = ExampleMatcher.matching()
        .withIgnoreCase()
        .withStringMatcher(ExampleMatcher.StringMatcher.CONTAINING);

Example<Product> example = Example.of(probe, matcher);

List<Product> result = productRepository.findAll(example);
```

Exemplu de `Specification`:

```
public static Specification<Product> hasMinPrice(BigDecimal minPrice) {
    return (root, query, cb) ->
                cb.greaterThanOrEqualTo(root.get("price"), minPrice);
}


public static Specification<Product> isActive() {
    return (root, query, cb) -> cb.isTrue(root.get("active"));
}

// utilizare:
productRepository.findAll(isActive()
                .and(hasMinPrice(new BigDecimal("1000"))));
```

2.14. Tranzacționalitatea și limitele repository-ului

În multe proiecte apare tentația de a considera că toate repository-urile sunt „automat tranzacționale” și că problema este astfel rezolvată. Realitatea este mai nuanțată. Operațiile standard moștenite primesc, în general, metadate tranzacționale rezonabile, însă metodele declarate explicit pot necesita `@Transactional`, în special când sunt combinate cu `@Modifying` sau când se dorește `readOnly=true`. [5, 6]

În proiectare, regula sănătoasă este ca tranzacția să fie stabilită la nivelul serviciului, acolo unde un caz de utilizare poate coordona mai multe repository-uri. Repository-ul poate avea metadate tranzacționale locale, dar frontiera conceptuală a tranzacției aparține, de regulă, stratului service.

De asemenea, trebuie înțeles rolul flush-ului. **EntityManager nu trimite neapărat instantaneu fiecare modificare către baza de date.** Sincronizarea efectivă are loc la `flush` sau la `commit`, ceea ce explică de ce anumite erori de constrângere pot apărea târziu, nu exact în momentul apelului `save(...)`.

> **Capcană frecventă**
>
> `saveAndFlush()` nu este un substitut pentru proiectarea corectă a tranzacțiilor. El forțează sincronizarea mai devreme, dar nu rezolvă problemele de modelare sau de ordonare logică a operațiilor.

2.15. Alte adnotări importante în Spring Data JPA

Adnotare	*Unde se aplică*	*Semnificație*
`@EnableJpaRepositories`	Configurație	Activează scanarea și infrastructura repository-urilor JPA.
`@Repository`	Clasă custom	Marchează componenta de persistență și activează traducerea excepțiilor Spring.
`@Transactional`	Servicii / repository	Definește frontiera tranzacțională și atribute precum readOnly.
`@Query`	Metodă repository	Declară explicit query-ul JPQL sau SQL nativ.
`@Modifying`	Metodă repository	Indică execuția unui update sau delete.
`@Param`	Parametru metodă	Leagă explicit argumentul metodei la parametrul numit din query.
`@EntityGraph`	Metodă repository	Controlează încărcarea relațiilor și reduce problemele de tip N+1.
`@Lock`	Metodă repository	Aplică un mod de locking JPA la executarea query-ului.
`@CreatedDate` `@LastModifiedDate`	Câmp entitate	Metadate de auditing pentru momentele de creare și modificare.
`@CreatedBy` `@LastModifiedBy`	Câmp entitate	Metadate de auditing pentru utilizatorul responsabil.

2.16. Auditing

Auditing-ul permite înregistrarea transparentă a informațiilor despre cine a creat sau a modificat o entitate și când s-a produs schimbarea. Pentru aceasta se activează infrastructura de auditing și se marchează câmpurile relevante cu adnotări dedicate. [5]

Dacă se urmăresc doar timestamp-urile, aplicația poate folosi `@CreatedDate` și `@LastModifiedDate`. Dacă se dorește și identificarea utilizatorului, este necesară de regulă implementarea unui bean `AuditorAware` care furnizează principalul curent.

Exemplu de activare a auditing-ului:

```
@Configuration
@EnableJpaAuditing
class PersistenceConfig {
}

@Entity
@EntityListeners(AuditingEntityListener.class)
public class Product {

    @CreatedDate
    private Instant createdAt;

    @LastModifiedDate
    private Instant updatedAt;
}
```

2.17. Repository-uri custom și fragmente

Nu toate cerințele pot fi exprimate prin query derivation, `@Query` sau Specification. Pentru logică de persistență specializată – de exemplu interogări JDBC foarte optimizate, apeluri către proceduri stocate sau compoziții neobișnuite – Spring Data permite atașarea de fragmente custom la repository. [5]

Modelul modern recomandat este fragment-based: se declară o interfață suplimentară cu metodele custom și o clasă de implementare pentru acel fragment. Apoi repository-ul principal extinde și acel fragment. În acest fel, codul specializat rămâne explicit și testabil.

Exemplu de fragment custom de repository:

```
public interface ProductRepositoryCustom {
    List<Product> findExpensiveActiveProducts();
}

public class ProductRepositoryCustomImpl implements ProductRepositoryCustom
{
    @PersistenceContext
    private EntityManager entityManager;

    @Override
    public List<Product> findExpensiveActiveProducts() {
```

```
        return entityManager.createQuery(
        "select p from Product p where p.active = true and p.price > 5000",
        Product.class
        ).getResultList();
    }
}

public interface ProductRepository extends JpaRepository<Product, Long>,
ProductRepositoryCustom {
}
```

2.18. @ConfigurationProperties pentru configurare structurată

În aplicațiile reale, proprietățile de configurare nu ar trebui dispersate prin întregul cod sub formă de string-uri injectate cu `@Value`. O soluție mult mai robustă este maparea lor într-o clasă tipizată prin `@ConfigurationProperties`.

Această abordare este relevantă și pentru configurațiile legate de persistență: dimensiuni implicite de pagină, limite administrative, ferestre temporale pentru batch-uri, toggles pentru seed data etc. În plus, asemenea clase pot fi marcate și cu `@Profile` atunci când este necesară activarea lor doar în anumite medii. [1]

Exemplu de proprietăți tipizate:

```
@ConfigurationProperties(prefix = "app.catalog")
public class CatalogProperties {

    private int defaultPageSize = 20;
    private int maxPageSize = 100;

    // getteri și setteri
}
```

Configurare în application.properties:

```
app.catalog.default-page-size=24
app.catalog.max-page-size=200
```

2.18.1. Probleme frecvente și bune practici

- Nu se pune logica de business în repository; ea aparține serviciului.
- Nu se expun entitățile JPA în toate straturile fără discernământ; pentru API-uri publice sunt adesea preferabile DTO-uri și proiecții.
- Nu se folosește query derivation pentru metode cu nume prea lung; în acel punct se trece la `@Query` sau Specification.
- Nu se activează open-in-view fără a înțelege consecințele asupra sesiunii și a încărcării de tip lazy.
- Nu se păstrează credențiale în clar în `application.properties` pentru producție; acestea se externalizează.

- Nu se supralicitează `saveAndFlush()`; se proiectează tranzacția și modelul de date corect.
- Se păstrează configurația comună în `application.properties` și diferențele în fișierele pe profil.
- Se preferă teste de repository și teste de integrare pentru validarea query-urilor, mapărilor și tranzacțiilor.

3. Aplicația care va fi realizată

Se va continua dezvoltarea aplicației pentru gestionarea produselor. Fiecare produs va avea:

- un identificator;
- un nume;
- un preț;
- o cantitate în stoc.

Aplicația va oferi în continuare operațiile:

- afișarea tuturor produselor;
- afișarea unui produs după identificator;
- adăugarea unui produs nou;
- modificarea unui produs existent;
- ștergerea unui produs.

Diferența față de laboratorul anterior este că, în loc ca datele să fie păstrate într-un Map sau într-o listă în memorie, ele vor fi salvate într-o bază de date MySQL.

4. Ce se schimbă față de laboratorul anterior

În aplicația anterioară exista un service care gestiona manual o colecție în memorie. Acum vom înlocui acel mecanism cu un strat de persistență real.

Evoluția naturală este următoarea:

1. obiectul Product devine entitate persistentă;
2. apare un Repository care se ocupă de operațiile de acces la date;
3. service-ul nu mai lucrează cu ConcurrentHashMap, ci cu repository-ul;
4. controller-ul rămâne aproape neschimbat ca responsabilitate;
5. configurația aplicației trebuie completată cu setările pentru baza de date.

Această tranziție este importantă: controller-ul continuă să expună aceeași interfață HTTP, dar mecanismul intern prin care sunt citite și salvate datele se schimbă.

5. Dependențele necesare

Pentru acest laborator, proiectul Spring Boot trebuie să conțină cel puțin următoarele dependențe:

- Spring Web
- Spring Data JPA
- MySQL Driver

În fișierul *pom.xml* vor apărea dependențele:

```
<dependencies>

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

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

5.1. Rolul dependenței Spring Data JPA

spring-boot-starter-data-jpa aduce infrastructura necesară pentru lucrul cu persistența relațională prin JPA și Hibernate. În acest context, ea oferă:

- maparea claselor Java către tabele;
- maparea obiectelor către înregistrări;
- operații CRUD gata implementate prin repository-uri;
- integrare cu tranzacții și cu sursa de date configurată.

Cel mai vizibil avantaj este că nu trebuie scrise manual interogări SQL pentru operațiile de bază. [1, 5]

6. Baza de date: două variante de lucru

Pentru laborator sunt propuse două moduri de rulare a bazei de date:

- varianta A: baza de date rulează în **Docker**;
- varianta B: baza de date este instalată **standalone** pe sistem.

Docker este foarte comod pentru laborator și pentru lucru reproductibil, iar instalarea standalone este utilă atunci când se dorește o bază de date rulată direct pe calculator, fără containere.

6.1. Varianta A: MySQL în Docker

Pentru MySQL, comanda este:

```
docker run --name lab-mysql \
  -e MYSQL_ROOT_PASSWORD=rootpass \
  -e MYSQL_DATABASE=rest_lab_db \
  -e MYSQL_USER=labuser \
  -e MYSQL_PASSWORD=labpass \
  -p 3306:3306 \
  -d mysql:latest
```

6.1.1. Verificarea containerului

Pentru a verifica dacă baza de date rulează:

```
docker ps
```

Pentru a vedea logurile:

```
docker logs lab-mysql
```

Este important ca în loguri să apară faptul că serverul este pornit și acceptă conexiuni.

6.1.2. Varianta B: instalare standalone

În această variantă MySQL este instalat direct în sistemul de operare.

După instalare, pentru laborator este necesar:

1. să existe serverul pornit;
2. să existe o bază de date dedicată;
3. să existe un utilizator care are drepturi asupra acelei baze de date.

Un exemplu minimal de configurare SQL este:

```
CREATE DATABASE rest_lab_db;
CREATE USER 'labuser'@'localhost' IDENTIFIED BY 'labpass';
GRANT ALL PRIVILEGES ON rest_lab_db.* TO 'labuser'@'localhost';
FLUSH PRIVILEGES;
```

În multe cazuri, această configurare poate fi realizată și dintr-un client grafic, precum:

- MySQL Workbench;
- **DBeaver**;
- HeidiSQL;
- phpMyAdmin.

7. Configurarea aplicației Spring Boot

Odată ce baza de date există și rulează, aplicația trebuie configurată să se conecteze la ea.

Acest lucru se face în `application.properties` sau `application.yml`.

7.1. Varianta application.properties pentru MySQL

```
spring.datasource.url=jdbc:mysql://localhost:3306/rest_lab_db?useSSL=false&serverTimezone=UTC
spring.datasource.username=labuser
spring.datasource.password=labpass
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
```

7.2. Varianta application.yml pentru MySQL

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/rest_lab_db?useSSL=false&serverTimezone=UTC
    username: labuser
    password: labpass
    driver-class-name: com.mysql.cj.jdbc.Driver

  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    open-in-view: false
    properties:
      hibernate:
        format_sql: true
    database-platform: org.hibernate.dialect.MySQLDialect
```

7.3. Semnificația setărilor importante

Aceasta este una dintre cele mai importante secțiuni ale laboratorului, deoarece aici se stabilesc legăturile dintre aplicație, JDBC, Hibernate și baza de date.

7.3.1. spring.datasource.url

Aceasta este adresa JDBC prin care aplicația se conectează la baza de date.

Exemplu:

```
spring.datasource.url=jdbc:mysql://localhost:3306/rest_lab_db?useSSL=false&serverTimezone=UTC
```

Elementele importante sunt:

- protocolul JDBC: `jdbc:mysql:`
- host-ul: `localhost`
- portul: `3306`
- numele bazei de date: `rest_lab_db`

Dacă baza de date rulează în Docker pe alt port, acel port trebuie folosit aici.

7.3.2. spring.datasource.username și spring.datasource.password

Aceste proprietăți definesc credențialele folosite de aplicație pentru conectare. Este recomandat ca aplicația să folosească un cont dedicat, nu contul de administrare root.

7.3.3. spring.datasource.driver-class-name

Specifică explicit clasa driver-ului JDBC. În multe situații Spring o poate deduce automat, dar în laborator este util să fie declarată clar.

Exemplu:

```
com.mysql.cj.jdbc.Driver
```

7.3.4. spring.jpa.hibernate.ddl-auto

Aceasta este una dintre cele mai importante setări din laborator. Ea controlează modul în care Hibernate gestionează schema bazei de date.

Valori uzuale:

- `none` – nu face nimic asupra schemei;
- `validate` – verifică dacă schema corespunde mapărilor;
- `update` – încearcă să actualizeze schema;
- `create` – recreează schema la pornire;
- `create-drop` – creează schema la pornire și o șterge la oprire.

Pentru laborator, o alegere practică este:

```
spring.jpa.hibernate.ddl-auto=update
```

Asta permite ca tabelele să fie create sau ajustate automat pe baza entităților.

7.3.5. spring.jpa.show-sql=true

Afișează în consolă instrucțiunile SQL generate de Hibernate. Pentru învățare este foarte util, deoarece studentul poate observa ce interogări sunt executate efectiv.

7.3.6. spring.jpa.properties.hibernate.format_sql=true

Formatează SQL-ul afișat, astfel încât să fie mai ușor de citit.

7.3.7. spring.jpa.open-in-view=false

Această setare controlează dacă contextul de persistență rămâne deschis și în timpul construirii răspunsului web. Într-o aplicație de laborator simplă nu pare foarte vizibilă, dar este o setare bună de controlat.

În acest laborator vom considera:

```
spring.jpa.open-in-view=false
```

astfel încât granițele dintre accesul la date și stratul web să fie mai clare.

7.3.8. spring.jpa.database-platform

Specifică dialectul **Hibernate** folosit pentru baza de date. Hibernate adaptează SQL-ul generat în funcție de acest dialect.

Exemplu:

```
org.hibernate.dialect.MySQLDialect
```

8. Structura proiectului

După introducerea persistenței, o structură potrivită este:

```
src/main/java/ro/unitbv/restlab
│
├── RestLabApplication.java       # Clasa principală (Entry Point)
│
├── controller
│   └── ProductController.java    # Expune endpoint-urile REST (API)
│
├── service
│   └── ProductService.java       # Logica de business a aplicației
│
├── repository
│   └── ProductRepository.java    # Interfața pentru baza de date (JPA)
│
├── model
│   └── Product.java              # Entitatea mapată pe baza de date
│
└── dto (Data Transfer Object)
    └── CreateProductRequest.java # Obiectul pentru validarea input-ului
```

9. Transformarea clasei Product într-o entitate persistentă

În laboratorul anterior, Product era un simplu model Java. Acum clasa trebuie să fie mapată către un tabel. [6]

9.1. Clasa Product

```
package ro.unitbv.restlab.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
```

```
    private double price;
    private int stock;

    public Product() {
    }

    public Product(Integer id, String name, double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Product(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

9.1.1. Ce face @Entity

Adnotarea `@Entity` marchează clasa ca entitate JPA. Cu alte cuvinte, ea spune framework-ului că obiectele de tip Product trebuie tratate ca date persistente.

În mod normal, o entitate va fi mapată către un tabel.

9.1.2. Ce face @Table(name = "products")

Această adnotare specifică explicit numele tabelului din baza de date. În acest caz, entitatea `Product` va fi mapată la tabelul `products`.

Dacă această adnotare nu ar exista, Hibernate ar alege un nume implicit.

9.1.3. Ce face @Id

`@Id` marchează câmpul care reprezintă cheia primară a entității.

În cazul nostru:

```
private Integer id;
```

este identificatorul unic al fiecărui produs.

9.1.4. Ce face @GeneratedValue(strategy = GenerationType.IDENTITY)

Această adnotare spune că valoarea cheii primare este generată de baza de date. Strategia `IDENTITY` este potrivită pentru coloane auto-increment în MySQL.

Practic:

- aplicația creează un produs fără să îi seteze manual id;
- la inserare, baza de date generează automat valoarea;
- Hibernate recuperează acel identificator și îl setează în obiect.

Acesta este motivul pentru care, după introducerea bazei de date, nu mai este potrivit ca clientul să trimită el însuși id la creare.

10. Introducerea repository-ului

După ce există o entitate, următorul pas firesc este definirea unui repository.

10.1. Interfața ProductRepository

```
package ro.unitbv.restlab.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import ro.unitbv.restlab.model.Product;

public interface ProductRepository extends JpaRepository<Product, Integer>
{
}
```

10.2. Ce este un repository

Un repository este componenta prin care aplicația accesează datele persistente. În acest laborator, repository-ul ascunde detaliile tehnice ale interacțiunii cu baza de date. [5]

Prin extinderea lui JpaRepository<Product, Integer>, interfața primește automat metode precum:

- findAll()
- findById(...)
- save(...)
- deleteById(...)
- existsById(...)

Aici apare unul dintre cele mai mari avantaje ale Spring Data: pentru operațiile CRUD simple nu mai este nevoie să fie scris cod SQL sau JDBC manual.

11. Adaptarea DTO-ului de creare

Fișier: `dto/CreateProductRequest.java`

```
package ro.unitbv.restlab.dto;

public class CreateProductRequest {

    private String name;
    private double price;
    private int stock;

    public CreateProductRequest() { }

    public CreateProductRequest(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
```

```
    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

Acest DTO rămâne foarte potrivit și în noua variantă, deoarece clientul nu trebuie să controleze cheia primară generată de baza de date.

12. Refactorizarea service-ului pentru baza de date

12.1. ProductService

```
package ro.unitbv.restlab.service;

import org.springframework.stereotype.Service;
import ro.unitbv.restlab.dto.CreateProductRequest;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;

import java.util.List;

@Service
public class ProductService {

    private final ProductRepository productRepository;

    public ProductService(ProductRepository productRepository) {
        this.productRepository = productRepository;
    }



    public List<Product> findAll() {
        return productRepository.findAll();
    }

    public Product findById(Integer id) {
        return productRepository.findById(id).orElse(null);
    }

    public Product create(CreateProductRequest request) {
        Product product = new Product(
                                request.getName(),
                                request.getPrice(),
                                request.getStock());
        return productRepository.save(product);
    }
```

```
    public Product update(Integer id, Product productData) {
        Product existingProduct = productRepository.findById(id)
                                                   .orElse(null);

        if (existingProduct == null) {
            return null;
        }

        existingProduct.setName(productData.getName());
        existingProduct.setPrice(productData.getPrice());
        existingProduct.setStock(productData.getStock());

        return productRepository.save(existingProduct);
    }

    public boolean deleteById(Integer id) {
        if (!productRepository.existsById(id)) {
            return false;
        }

        productRepository.deleteById(id);
        return true;
    }
}
```

12.2. Evoluția față de laboratorul anterior

Diferența centrală este clară:

- înainte, service-ul lucra direct cu un Map;
- acum, service-ul delegă operațiile de persistență către ProductRepository.

Această modificare schimbă mecanismul intern, dar nu schimbă responsabilitatea generală a service-ului: el rămâne locul în care se coordonează operațiile asupra produselor.

13. Controller-ul adaptat

Controller-ul poate rămâne aproape identic, ceea ce este un semnal foarte bun: interfața HTTP a aplicației nu depinde direct de detaliile stocării.

13.1. ProductController

```
package ro.unitbv.restlab.controller;

import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.CreateProductRequest;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.service.ProductService;

import java.util.List;
```

```
@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public List<Product> getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(
                                            @PathVariable Integer id) {
        Product product = productService.findById(id);

        if (product == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(product);
    }

    @PostMapping
    public ResponseEntity<Product> createProduct(
                             @RequestBody CreateProductRequest request) {
        Product createdProduct = productService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED)
                            .body(createdProduct);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(
                                            @PathVariable Integer id,
                                            @RequestBody Product product) {
        Product updatedProduct = productService.update(id, product);

        if (updatedProduct == null) {
            return ResponseEntity.notFound().build();
        }

        return ResponseEntity.ok(updatedProduct);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        boolean deleted = productService.deleteById(id);


        if (!deleted) {
            return ResponseEntity.notFound().build();
```

```
            }

            return ResponseEntity.noContent().build();
        }
    }
```

14. Clasa principală

Fișier: `RestLabApplication.java`

```
package ro.unitbv.restlab;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestLabApplication {

    public static void main(String[] args) {
        SpringApplication.run(RestLabApplication.class, args);
    }
}
```

15. Ce se întâmplă la pornirea aplicației

Dacă setarea:

```
spring.jpa.hibernate.ddl-auto=update
```

este activă, atunci la pornire Hibernate va analiza entitățile și va crea sau actualiza schema bazei de date. [5]

În cazul nostru, va apărea un tabel `products`, cu coloane corespunzătoare câmpurilor entității.

Dacă `show-sql=true`, în consolă vor apărea instrucțiunile SQL generate.

16. Testarea aplicației

16.1. Crearea unui produs

```
POST /api/products
Content-Type: application/json
Body:
{
  "name": "Monitor",
  "price": 950.0,
  "stock": 7
}
```

Observație importantă: `id` nu mai este trimis, deoarece el este generat automat de baza de date.

16.2. Listarea produselor

```
GET /api/products
```

Răspunsul va conține produsele salvate în baza de date.

16.3. Căutarea după id

```
GET /api/products/1
```

16.4. Actualizarea

```
PUT /api/products/1
Content-Type: application/json
Body:
{
  "id": 999,
  "name": "Monitor 27 inch",
  "price": 1200.0,
  "stock": 5
}
```

În implementarea noastră, `id` din URL este reperul operației, iar service-ul actualizează obiectul existent.

16.5. Ștergerea

```
DELETE /api/products/1
```

17. Popularea inițială a bazei de date

Pentru laborator, uneori este util ca aplicația să pornească deja cu câteva date. Acest lucru se poate face cu un fișier `data.sql`.

Fișier: `src/main/resources/data.sql`

```
INSERT INTO products (name, price, stock) VALUES ('Laptop', 3500.0, 10);
INSERT INTO products (name, price, stock) VALUES ('Mouse', 120.0, 50);
INSERT INTO products (name, price, stock) VALUES ('Keyboard', 250.0, 30);
```

În acest mod, la pornirea aplicației pot fi introduse automat date de test, dacă schema este deja disponibilă.

18. Observații privind setările și mediile de lucru

Într-o aplicație reală, este foarte util să existe profiluri distincte, de exemplu:

- `dev` pentru dezvoltare locală;
- `docker` pentru rulare cu container;
- `prod` pentru producție.

Pentru laborator, această idee poate fi introdusă simplu prin fișiere separate:

- application.properties
- application-docker.properties

Această organizare devine foarte utilă atunci când aceeași aplicație trebuie să ruleze în mai multe contexte. [1]

19. Ce a adus Spring Data în mod concret

După parcurgerea laboratorului, avantajele introducerii Spring Data sunt vizibile:

- obiectele aplicației sunt persistate în baza de date;
- nu mai este nevoie de colecții manuale în memorie;
- operațiile CRUD de bază sunt obținute rapid prin JpaRepository;
- modelul aplicației devine mai apropiat de o aplicație enterprise reală;
- controller-ul poate rămâne concentrat pe interfața HTTP;
- service-ul poate rămâne concentrat pe logica aplicației;
- repository-ul preia responsabilitatea accesului la date.

Această organizare continuă foarte firesc modelul pe straturi introdus în laboratorul anterior.

20. Întrebări recapitulative

1. Ce rol are dependența spring-boot-starter-data-jpa?
2. Ce reprezintă o entitate JPA?
3. Ce rol are adnotarea `@Entity`?
4. Ce rol are `@Id`?
5. Ce rol are `@GeneratedValue(strategy = GenerationType.IDENTITY)`?
6. Ce este un JpaRepository?
7. Ce metodă din repository este folosită pentru salvare?
8. Ce proprietate configurează URL-ul de conectare la baza de date?
9. Ce efect are `spring.jpa.hibernate.ddl-auto=update`?
10. De ce este util ca `id` să fie generat de baza de date?
11. Care este diferența practică dintre rularea bazei de date în Docker și instalarea standalone?
12. De ce controller-ul poate rămâne aproape neschimbat, deși mecanismul de stocare s-a modificat complet?

21. Exerciții propuse

21.1. Exercițiul 1

Adăugați în `ProductRepository` o metodă pentru căutarea produselor după nume.

Indiciu:

```
List<Product> findByNameContainingIgnoreCase(String name);
```

Apoi expuneți un endpoint de forma:

```
GET /api/products/search?name=lap
```

21.2. Exercițiul 2

Adăugați o metodă pentru găsirea produselor al căror preț este mai mic decât o anumită valoare.

Laborator 3: Elevarea aplicației REST – DTO-uri, validări și tratare centralizată a erorilor

1. Ce se urmărește în această etapă

În forma actuală, aplicația funcționează corect: expune endpoint-uri REST, folosește Spring Data JPA și salvează produsele într-o bază de date. Totuși, pe măsură ce codul este privit mai atent, apar câteva observații naturale.

În varianta precedentă:

- controller-ul primește și returnează direct entitatea `Product` în unele situații;
- validările sunt puține sau inexistente;
- erorile sunt tratate local, direct în metodele controller-ului;
- modelul intern al aplicației și contractul API sunt încă prea apropiate.

În această etapă, aplicația va fi reorganizată astfel încât:

1. request-urile venite de la client să fie reprezentate prin DTO-uri dedicate;
2. response-urile trimise clientului să fie reprezentate prin DTO-uri dedicate;
3. datele de intrare să fie validate automat;
4. erorile să fie tratate centralizat, într-un singur loc;
5. fluxul controller → service → repository să devină mai clar.

Această variantă este foarte potrivită pentru a arăta cum evoluează o aplicație de la o implementare funcțională la una mai bine structurată.

2. De ce apar DTO-uri separate pentru request și response

Până acum, aplicația a folosit în mai multe locuri aceeași clasă sau structuri foarte apropiate. Inițial, această abordare s-a dovedit practică, însă, în prezent, se constată că este necesară o delimitare mai clară a rolurilor obiectelor. [38, 39]

Apare o întrebare simplă: datele pe care clientul le trimite la creare sunt identice cu datele pe care serverul le trimite înapoi? De multe ori, nu sunt.

De exemplu:

- la creare, clientul nu trebuie să trimită `id`;
- la răspuns, serverul trebuie să trimită `id`;
- în viitor, unele câmpuri ar putea fi expuse doar la citire;
- anumite detalii interne ar putea să nu fie niciodată expuse în API.

Prin urmare, este util să avem:

- un DTO pentru creare;
- un DTO pentru actualizare;
- un DTO pentru răspuns.

Această separare face contractul API mai clar și reduce cuplajul dintre modelul intern și interfața externă.

3. Structura proiectului în noua variantă

O structură curată poate fi:

```
src/main/java/ro/unitbv/restlab
│
├── RestLabApplication.java
├── controller
│   └── ProductController.java
├── dto
│   ├── request
│   │   ├── CreateProductRequest.java
│   │   └── UpdateProductRequest.java
│   └── response
│       ├── ErrorResponse.java
│       └── ProductResponse.java
├── exception
│   ├── GlobalExceptionHandler.java
│   ├── ProductNotFoundException.java
│   └── ValidationExceptionResponse.java
├── model
│   └── Product.java
├── repository
│   └── ProductRepository.java
├── service
│   └── ProductService.java
└── mapper
    └── ProductMapper.java
```

Această structură păstrează ideile deja cunoscute și introduce câteva elemente noi exact acolo unde sunt utile.

4. Entitatea rămâne modelul persistent

Entitatea `Product` rămâne responsabilă pentru maparea către baza de date.

4.1. Product.java

```
package ro.unitbv.restlab.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
```

```
import jakarta.persistence.Table;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private double price;
    private int stock;

    public Product() {
    }

    public Product(Integer id, String name, double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Product(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }
```

```
    public int getStock() {
        return stock;
    }


    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

> **Observația importantă**
>
> Entitatea nu mai trebuie privită ca obiect de transport HTTP. Ea rămâne modelul persistent, adică forma internă cu care aplicația lucrează în relația cu baza de date.

5. DTO-uri pentru request

5.1. DTO pentru creare

Fișier: `dto/request/CreateProductRequest.java`

```
package ro.unitbv.restlab.dto.request;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class CreateProductRequest {

    @NotBlank(message = "Numele produsului este obligatoriu.")
    @Size(max = 100,
          message = "Numele produsului nu poate depăși 100 de caractere.")
    private String name;

    @DecimalMin(value = "0.01",
                message = "Prețul trebuie să fie mai mare decât 0.")
    private double price;

    @Min(value = 0, message = "Stocul nu poate fi negativ.")
    private int stock;

    public CreateProductRequest() {
    }

    public CreateProductRequest(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

5.2. DTO pentru actualizare

Fișier: `dto/request/UpdateProductRequest.java`

```
package ro.unitbv.restlab.dto.request;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;


public class UpdateProductRequest {

    @NotBlank(message = "Numele produsului este obligatoriu.")
    @Size(max = 100,
          message = "Numele produsului nu poate depăși 100 de caractere.")
    private String name;

    @DecimalMin(value = "0.01",
                message = "Prețul trebuie să fie mai mare decât 0.")
    private double price;

    @Min(value = 0, message = "Stocul nu poate fi negativ.")
    private int stock;

    public UpdateProductRequest() {
    }
```

```
    public UpdateProductRequest(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

În această variantă, DTO-ul de actualizare seamănă cu cel de creare. Totuși, separarea este utilă deoarece cele două operații pot evolua diferit mai târziu. La început, poate părea puțin redundant, dar arhitectural este o alegere sănătoasă.

6. DTO pentru response

Fișier: `dto/response/ProductResponse.java`

```
package ro.unitbv.restlab.dto.response;

public class ProductResponse {

    private Integer id;
    private String name;
    private double price;
    private int stock;

    public ProductResponse() {
    }
```

```
    public ProductResponse(Integer id, String name,
                           double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

Acum API-ul nu mai expune direct entitatea `Product`, ci un DTO de răspuns. Această alegere este importantă deoarece păstrează o separare clară între modelul intern al aplicației și contractul public al API-ului. Cu alte cuvinte, clasa folosită intern pentru persistență și logică de aplicație nu mai este aceeași clasă pe care clientul o vede în răspunsurile HTTP. [38]

Acest lucru aduce mai multe avantaje practice.

În primul rând, aplicația câștigă în **flexibilitate**. Entitatea `Product` poate evolua intern fără a obliga la modificarea imediată a contractului extern. De exemplu, în viitor, entitatea ar putea conține și câmpuri suplimentare precum:

- `supplierCode`
- `internalNotes`
- `createdAt`
- `lastModifiedAt`
- `costPrice`

Unele dintre aceste câmpuri pot fi utile pentru aplicație sau pentru baza de date, dar nu este deloc sigur că ele trebuie trimise către client. Dacă API-ul ar expune direct entitatea, toate aceste modificări ar risca să afecteze și răspunsurile externe. Prin folosirea unui DTO de răspuns, aplicația poate decide explicit ce anume devine vizibil.

În al doilea rând, DTO-ul de răspuns ajută la **controlul informației expuse**. Nu tot ceea ce există în modelul intern trebuie să fie public. De exemplu, dacă produsul ar avea și un câmp `costPrice` care reprezintă costul intern de achiziție, acel câmp ar putea fi important pentru calcule interne, dar nu ar trebui trimis către frontend sau către consumatorii externi ai API-ului. DTO-ul de răspuns permite expunerea doar a câmpurilor relevante, de exemplu: `id`, `name`, `price`, `stock`, și nimic mai mult.

În al treilea rând, această separare ajută la **stabilitatea contractului API**. Clienții unui API se bazează pe structura răspunsurilor primite. Dacă acea structură depinde direct de entitatea JPA, atunci orice schimbare internă poate produce efecte nedorite asupra clientului. Un DTO de răspuns funcționează ca o formă de protecție: contractul extern rămâne stabil chiar dacă modelul intern este reorganizat.

Un exemplu aplicat foarte clar este următorul. Se presupune că, la început, entitatea și răspunsul au aceeași formă:

```
{
  "id": 1,
  "name": "Laptop",
  "price": 3500.0,
  "stock": 10
}
```

Mai târziu, aplicația evoluează și în entitate se adaugă câmpurile:

```
private String supplierCode;
private double costPrice;
private LocalDateTime createdAt;
```

Dacă API-ul expune direct entitatea, există riscul ca răspunsul să devină:

```
{
  "id": 1,
  "name": "Laptop",
  "price": 3500.0,
  "stock": 10,
  "supplierCode": "SUP-19",
  "costPrice": 2800.0,
  "createdAt": "2026-03-22T11:20:00"
}
```

Din perspectiva clientului, această schimbare poate fi nedorită sau chiar periculoasă. Cu un DTO de răspuns, aplicația poate continua să trimită exact forma dorită:

```
{
  "id": 1,
  "name": "Laptop",
  "price": 3500.0,
  "stock": 10
}
```

Un alt avantaj apare atunci când forma răspunsului trebuie adaptată pentru client, nu pentru baza de date. De exemplu, intern produsul poate avea două câmpuri separate:

```
private String brand;
private String model;
```

dar în răspunsul API poate fi mai util un singur câmp calculat:

```
{
  "id": 1,
  "displayName": "Lenovo ThinkPad T14",
  "price": 4200.0,
  "stock": 8
}
```

Aici DTO-ul de răspuns nu mai este doar o copie a entității, ci un obiect gândit special pentru consumul extern. Acesta este un caz foarte frecvent în aplicațiile reale.

DTO-urile de răspuns sunt utile și atunci când există mai multe tipuri de clienți. De exemplu:

- un endpoint pentru administrare poate întoarce mai multe detalii;
- un endpoint public poate întoarce doar informațiile esențiale;
- un endpoint pentru listare poate întoarce o formă scurtă;
- un endpoint pentru detalii poate întoarce o formă extinsă.

Astfel, pot exista DTO-uri diferite, precum:

- `ProductSummaryResponse`
- `ProductDetailResponse`
- `AdminProductResponse`

Toate pot proveni din aceeași entitate, dar fiecare este adaptat unui anumit context de utilizare.

Mai apare un avantaj important și în relația cu JPA: entitățile sunt obiecte persistente, legate de contextul de persistență și de mapările către baza de date. Ele nu sunt întotdeauna cele mai potrivite obiecte pentru a fi trimise direct în exterior. Menținerea lor în interiorul aplicației și conversia către DTO-uri de răspuns face arhitectura mai curată și reduce dependența stratului web de detaliile persistenței.

7. Bean Validation

Bean Validation reprezintă un mecanism standard (specificație) în ecosistemul Java (Jakarta Validation) prin care se pot defini reguli de validare direct asupra câmpurilor unui obiect, folosind adnotări. Aceste reguli sunt evaluate automat de framework (de exemplu Spring Boot) înainte ca logica aplicației să fie executată. [20]

Bean Validation (Jakarta Validation) este doar o specificație (standard), nu o implementare concretă. Asta înseamnă că doar:

- definește **API-ul** (adnotările: @NotNull, @Size, etc.);
- definește **comportamentul** (când și cum se validează);
- definește **contractul** (ce trebuie să facă o implementare).

Dar **nu conține codul efectiv care execută validarea**.

Pentru ca validarea să funcționeze, este necesară o **implementare concretă** a standardului.

Cea mai utilizată este:

- **Hibernate Validator** (implementarea de referință pentru Jakarta Validation)

În Spring Boot, lucrurile sunt automatizate; dependența `spring-boot-starter-validation` aduce automat:

- API-ul Jakarta Validation
- implementarea (Hibernate Validator)

Astfel:

- adnotările sunt definite de standard;
- validarea efectivă este realizată de Hibernate Validator;
- Spring doar integrează mecanismul în fluxul web (prin `@Valid`).

7.1. Mecanism general

În practică, validarea este declarată la nivelul claselor (de regulă DTO-uri de request), prin adnotări precum:

```
@NotBlank(message = "Numele este obligatoriu")
@Size(max = 100)
private String name;

@DecimalMin(value = "0.01", message = "Prețul trebuie să fie pozitiv")
private double price;
```

Atunci când un astfel de obiect este utilizat într-un controller împreună cu adnotarea `@Valid`:

```
public ResponseEntity<?> create(
                        @Valid @RequestBody CreateProductRequest request)
```

are loc următoarea succesiune de pași:

1. Deserializarea: conținutul JSON din request este convertit într-un obiect Java.
2. Validarea: sunt evaluate toate constrângerile definite prin adnotări.
3. Rezultatul validării:
   - dacă toate regulile sunt respectate, execuția metodei controller continuă normal;
   - dacă există erori, execuția metodei este oprită.

În cazul în care validarea eșuează, Spring aruncă automat o excepție de tip `MethodArgumentNotValidException`. [20]

Această excepție este ulterior tratată:

- fie de mecanismul implicit al framework-ului;
- fie de un `@RestControllerAdvice` definit în aplicație.

7.2. Comportamentul implicit (fără tratare personalizată)

Dacă nu există un handler global de excepții, Spring Boot returnează automat un răspuns HTTP:

```
400 Bad Request
```

cu un corp JSON standard, care conține informații despre eroare (structura exactă poate varia în funcție de versiune):

```
{
  "timestamp": "...",
  "status": 400,
  "error": "Bad Request",
  "path": "/api/products"
}
```

În mod implicit, detaliile despre fiecare câmp invalid nu sunt prezentate într-o formă foarte prietenoasă, motiv pentru care, în aplicațiile reale, se definește aproape întotdeauna un mecanism personalizat de tratare a erorilor.

Un aspect esențial al Bean Validation este faptul că:

- validarea are loc **înainte** de execuția metodei controller;
- dacă validarea eșuează, metoda controller **nu este apelată deloc**.

Acest lucru implică faptul că:

- nu este necesară verificarea manuală a câmpurilor în controller;
- nu se ajunge în service sau repository dacă datele sunt invalide;
- fluxul este întrerupt devreme, ceea ce simplifică logica aplicației.

7.3. Integrarea în fluxul aplicației

Fluxul complet într-o aplicație REST este:

1. clientul trimite un request JSON;
2. Spring deserializază JSON-ul într-un obiect DTO;
3. Bean Validation verifică obiectul:
    - dacă este valid → se continuă execuția;
    - dacă este invalid → se aruncă excepție;
4. (opțional) excepția este interceptată de un `@RestControllerAdvice`;
5. se construiește un răspuns HTTP adecvat (de regulă `400 Bad Request`).

7.4. Avantaje

Utilizarea Bean Validation oferă câteva beneficii clare:

- cod mai curat – validările nu mai sunt scrise manual;
- declarativitate – regulile sunt vizibile direct lângă date;
- execuție automată – nu este necesară apelarea explicită a validării;
- integrare nativă cu Spring – funcționează direct cu `@Valid`;
- separarea responsabilităților – validarea este separată de logica de business;
- oprirea timpurie a fluxului – datele invalide nu ajung în straturile inferioare.

7.5. Dependența necesară

Pentru a folosi validările din DTO-uri, este necesară și dependența corespunzătoare. În `pom.xml` trebuie să existe și:

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

Această dependență activează infrastructura Bean Validation în Spring Boot.

7.6. Principalele adnotări Bean Validation

Bean Validation oferă un set extins de adnotări care acoperă cele mai frecvente scenarii de validare. Mai jos sunt prezentate cele mai utilizate în aplicațiile REST. [20]

7.7. Adnotări pentru valori obligatorii

```
@NotNull
```

Verifică faptul că valoarea nu este `null`. Nu verifică dacă este șir vid (în cazul stringurilor).

```
@NotBlank
```

Se folosește pentru șiruri de caractere și verifică faptul că valoarea:

- nu este `null`;

- nu este vidă;
- nu conține doar spații.

Este potrivită pentru nume, titluri, descrieri.

```
@NotEmpty
```

Se aplică pentru colecții, array-uri sau stringuri și verifică faptul că:

- nu sunt `null`;
- nu sunt goale.

Diferența față de `@NotBlank`: nu elimină spațiile (pentru stringuri).

7.7.1. Adnotări pentru dimensiune

```
@Size(min = 2, max = 100)
```

Controlează dimensiunea unui string, array sau colecție.

Exemplu:

- pentru nume: max 100 caractere;
- pentru liste: număr minim/maxim de elemente.

7.7.2. Adnotări pentru valori numerice

```
@Min(value = 0)
```

Valoarea trebuie să fie ≥ valoarea specificată (pentru tipuri întregi).

```
@Max(value = 1000)
```

Valoarea trebuie să fie ≤ pragul specificat.

```
@DecimalMin(value = "0.01")
```

Valoarea trebuie să fie ≥ pragul specificat (pentru numere zecimale).

```
@DecimalMax(value = "10000.0")
```

Similar pentru limită superioară.

```
@Positive
```

Valoarea trebuie să fie strict pozitivă (> 0).

```
@PositiveOrZero
```

Valoarea trebuie să fie ≥ 0.

```
@Negative, @NegativeOrZero
```

Pentru cazurile inverse.

7.7.3. Adnotări pentru formate specifice

```
@Email
```

Verifică dacă stringul respectă formatul unui email.

```
@Pattern(regexp = "...")
```

Permite validarea pe baza unei expresii regulate.

Exemplu:

```
@Pattern(regexp = "^[A-Z0-9]+$")
```

7.7.4. Adnotări pentru date calendaristice

```
@Past
```

Valoarea trebuie să fie în trecut.

```
@Future
```

Valoarea trebuie să fie în viitor.

```
@PastOrPresent, @FutureOrPresent
```

Variante mai permisive.

În sinteză:

Categorie	***Adnotare***	***Se aplică pe***	***Descriere***	***Exemplu utilizare***
Obligatoriu	@NotNull	orice tip	Valoarea nu poate fi null	id, referințe
	@NotBlank	String	Nu este null, nu este gol, nu conține doar spații	nume produs
	@NotEmpty	String, colecții, array	Nu este null și nu este gol	listă elemente
Dimensiune	@Size(min, max)	String, colecții	Lungime sau număr de elemente în interval	nume max 100
Numeric – întregi	@Min	int, long	Valoare ≥ minim	stoc ≥ 0
	@Max	int, long	Valoare ≤ maxim	limită superioară
Numeric – zecimale	@DecimalMin	double, BigDecimal	Valoare ≥ minim (zecimal)	preț ≥ 0.01

Categorie	*Adnotare*	*Se aplică pe*	*Descriere*	*Exemplu utilizare*
	@DecimalMax	double, BigDecimal	Valoare ≤ maxim	plafon preț
Semn numeric	@Positive	numeric	> 0	preț
	@PositiveOrZero	numeric	≥ 0	stoc
	@Negative	numeric	< 0	valori negative
	@NegativeOrZero	numeric	≤ 0	cazuri speciale
Format	@Email	String	Format valid de email	email utilizator
	@Pattern	String	Validare cu regex	cod produs
Temporal	@Past	date/timp	Valoare în trecut	dată naștere
	@Future	date/timp	Valoare în viitor	dată expirare
	@PastOrPresent	date/timp	≤ prezent	timestamp
	@FutureOrPresent	date/timp	≥ prezent	programări
Altele utile	@AssertTrue	boolean	Trebuie să fie true	flag valid
	@AssertFalse	boolean	Trebuie să fie false	flag invalid

8. Mapper-ul dintre entitate și DTO-uri

Pe măsură ce aplicația evoluează și apar DTO-uri separate pentru request și response, devine necesar un mecanism clar prin care se realizează conversia între diferitele tipuri de obiecte. [38, 39]

Un **mapper** este o componentă responsabilă exclusiv pentru transformarea datelor dintr-un tip în altul, de exemplu:

- din DTO de request → în entitate;
- din entitate → în DTO de response;
- din DTO de update → în modificări asupra unei entități existente.

Cu alte cuvinte, mapper-ul acționează ca un „traducător” între straturi:

- stratul web (DTO-uri);
- stratul de business/persistență (entități).

8.1. De ce este necesar un mapper

Fără un mapper dedicat, conversiile ar ajunge să fie scrise direct în:

- controller (ceea ce îl face mai încărcat și mai greu de citit);
- service (unde ar amesteca logica de business cu detalii tehnice de conversie).

Centralizarea conversiilor într-o clasă separată aduce mai multe beneficii:

- cod mai curat – fiecare clasă are un rol bine definit;
- reutilizare – aceeași conversie poate fi folosită în mai multe locuri;
- mentenanță mai ușoară – modificările se fac într-un singur loc;
- lizibilitate mai bună – fluxul aplicației devine mai clar.

8.2. Bune practici în definirea mapper-elor

8.2.1. Responsabilitate unică

Un mapper ar trebui să conțină doar logică de conversie, fără:

- apeluri către repository;
- logică de business;
- validări complexe.

El trebuie să fie cât mai simplu și predictibil.

8.2.2. Conversii explicite

Este recomandat ca maparea să fie explicită, câmp cu câmp. Astfel, devine foarte clar:

- ce câmpuri sunt transferate;
- ce câmpuri sunt ignorate;
- ce transformări sunt aplicate.

8.2.3. Separarea tipurilor de conversii

Este util ca metodele să fie diferențiate clar în funcție de scop:

- `toEntity(...)` – pentru creare;
- `updateEntity(...)` – pentru modificarea unei entități existente;
- `toResponse(...)` – pentru expunerea datelor.

Această separare evită ambiguitățile și face codul mai expresiv.

8.2.4. Evitarea creării inutile de obiecte

În cazul update-urilor, este recomandat să se modifice entitatea existentă (de exemplu printr-o metodă `updateEntity`), nu să se creeze un obiect nou. Acest lucru este important mai ales în context JPA, unde entitățile sunt gestionate de contextul de persistență. [39]

8.2.5. Izolarea conversiilor într-un pachet dedicat

Mapper-ele sunt de obicei grupate într-un pachet separat (`mapper`), pentru a reflecta rolul lor transversal între straturi.

8.2.6. Adnotarea cu @Component

Prin adnotarea mapper-ului cu `@Component`, acesta devine un bean Spring și poate fi injectat în service. Astfel, conversiile sunt reutilizabile și integrate natural în aplicație.

8.2.7. Mapper: static vs componentă Spring

Există două stiluri principale de implementare a mapper-elor:

8.2.8. Mapper cu metode statice

```
public class ProductMapper {

  public static Product toEntity(CreateProductRequest request) {
        return new Product(
                request.getName(),
                request.getPrice(),
                request.getStock()
        );
    }
}
```

Avantaje

- simplu și direct – nu necesită injectare;
- nu depinde de Spring (poate fi folosit și în alte contexte);
- performanță ușor mai bună (nu există bean management, deși impactul este neglijabil);
- potrivit pentru mapări triviale (câmp → câmp).

Dezavantaje

- nu poate avea dependențe (ex: alt mapper, servicii, conversii complexe);
- mai greu de extins dacă logica devine mai complexă;
- mai dificil de mock-uit în teste (nu se poate injecta ușor).

8.2.9. Mapper ca @Component (bean Spring)

```
@Component
public class ProductMapper {
    ...
}
```

Avantaje

- poate fi injectat în service-uri;
- poate folosi alte componente (ex: alte mappere);
- este ușor de testat și mock-uit;
- permite evoluția către mapări mai complexe;
- este compatibil cu abordări precum:
  - MapStruct[9]
  - layering mai strict

Dezavantaje

- introduce o dependență de Spring;

[9] MapStruct este o bibliotecă Java care generează automat, la compilare, codul de mapare între obiecte (de exemplu între DTO-uri și entități), eliminând necesitatea scrierii manuale a mapper-elor.

- necesită injectare (puțin mai mult boilerplate[10]);
- poate părea suprainginerie pentru mapări foarte simple.

8.3. Recomandare practică

Se pot folosi metode statice când:

- maparea este trivială (1:1);
- nu există dependențe;
- aplicația este mică sau de tip utilitar;
- nu se urmărește testare avansată.

Se recomandă `@Component` când:

- aplicația este pe Spring;
- există mai multe mappere;
- mapările devin mai complexe;
- există logică de conversie (ex: formatări, agregări);
- se dorește testare și mock-uire.

În aplicațiile enterprise moderne, mapper-ele:

- sunt aproape întotdeauna componente injectabile, sau
- sunt generate automat (ex: `MapStruct`, care tot bean Spring devine).

8.4. Mapper-ul pentru Product

În acest moment apare nevoia unui loc clar în care se fac conversiile dintre tipuri. Nu este bine ca această logică să fie împrăștiată în controller sau service.

Fișier: `mapper/ProductMapper.java`

```
package ro.unitbv.restlab.mapper;

import org.springframework.stereotype.Component;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.model.Product;

@Component
public class ProductMapper {
    public Product toEntity(CreateProductRequest request) {
        return new Product(
                request.getName(),
                request.getPrice(),
                request.getStock()
        );
    }
```

[10] Boilerplate reprezintă cod standard, repetitiv și previzibil, care trebuie scris pentru funcționarea aplicației, dar care nu aduce logică specifică sau valoare directă problemei rezolvate.

```
    public void updateEntity(Product product,
                             UpdateProductRequest request) {
        product.setName(request.getName());
        product.setPrice(request.getPrice());
        product.setStock(request.getStock());
    }

    public ProductResponse toResponse(Product product) {
        return new ProductResponse(
                product.getId(),
                product.getName(),
                product.getPrice(),
                product.getStock()
        );
    }
}
```

Acest mapper face două lucruri importante:

- transformă request-urile în entități sau modificări de entități;
- transformă entitățile în response-uri.

9. Repository-ul rămâne simplu

Fișier: `repository/ProductRepository.java`

```
package ro.unitbv.restlab.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import ro.unitbv.restlab.model.Product;

public interface ProductRepository extends JpaRepository<Product, Integer>
{
}
```

Repository-ul nu trebuie complicat. El rămâne componenta care oferă operațiile de bază asupra datelor persistente. [38]

10. Excepție dedicată pentru resursa inexistentă

Până acum, pentru produs inexistent s-a returnat `null` din service, iar controller-ul a decis ce face mai departe. O variantă mai curată este ca service-ul să exprime direct problema printr-o excepție specifică.

Fișier: `exception/ProductNotFoundException.java`

```
package ro.unitbv.restlab.exception;

public class ProductNotFoundException extends RuntimeException {
```

```
    public ProductNotFoundException(Integer id) {
        super("Produsul cu id-ul " + id + " nu a fost găsit.");
    }
}
```

Această excepție exprimă clar problema de domeniu sau de aplicație: resursa cerută nu există.

11. DTO-uri pentru răspunsurile de eroare

11.1. Răspuns simplu de eroare

Fișier: `dto/response/ErrorResponse.java`

```
package ro.unitbv.restlab.dto.response;

import java.time.LocalDateTime;

public class ErrorResponse {

    private LocalDateTime timestamp;
    private int status;
    private String error;
    private String message;
    private String path;

    public ErrorResponse() { }

    public ErrorResponse(LocalDateTime timestamp, int status,
                         String error, String message, String path) {
        this.timestamp = timestamp;
        this.status = status;
        this.error = error;
        this.message = message;
        this.path = path;
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }
```

```
    public String getError() {
        return error;
    }

    public void setError(String error) {
        this.error = error;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getPath() {
        return path;
    }

    public void setPath(String path) {
        this.path = path;
    }
}
```

11.2. Răspuns pentru erori de validare

Fișier: `exception/ValidationExceptionResponse.java`

```
package ro.unitbv.restlab.exception;

import java.time.LocalDateTime;
import java.util.Map;

public class ValidationExceptionResponse {

    private LocalDateTime timestamp;
    private int status;
    private String error;
    private Map<String, String> validationErrors;
    private String path;

    public ValidationExceptionResponse() { }

    public ValidationExceptionResponse(LocalDateTime timestamp,
                                       int status,
                                       String error,
                                       Map<String, String> validationErrors,
                                       String path) {
        this.timestamp = timestamp;
        this.status = status;
        this.error = error;
        this.validationErrors = validationErrors;
```

```
        this.path = path;
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }

    public String getError() {
        return error;
    }

    public void setError(String error) {
        this.error = error;
    }

    public Map<String, String> getValidationErrors() {
        return validationErrors;
    }

    public void setValidationErrors(Map<String, String> validationErrors) {
        this.validationErrors = validationErrors;
    }

    public String getPath() {
        return path;
    }

    public void setPath(String path) {
        this.path = path;
    }
}
```

Se remarcă faptul că structura erorilor de validare diferă de cea a erorilor generale, întrucât este recomandat ca fiecare câmp invalid și mesajul aferent să fie transmise separat.

12. Service-ul devine mai expresiv

Fișier: `service/ProductService.java`

```
package ro.unitbv.restlab.service;

import org.springframework.stereotype.Service;
```

```
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.mapper.ProductMapper;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;

import java.util.List;

@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    public ProductService(ProductRepository productRepository,
                          ProductMapper productMapper) {
        this.productRepository = productRepository;
        this.productMapper = productMapper;
    }

    public List<ProductResponse> findAll() {
        return productRepository.findAll()
                .stream()
                .map(productMapper::toResponse)
                .toList();
    }

    public ProductResponse findById(Integer id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ProductNotFoundException(id));

        return productMapper.toResponse(product);
    }

    public ProductResponse create(CreateProductRequest request) {
        Product product = productMapper.toEntity(request);
        Product savedProduct = productRepository.save(product);
        return productMapper.toResponse(savedProduct);
    }

    public ProductResponse update(Integer id,
                                  UpdateProductRequest request) {
        Product existingProduct = productRepository.findById(id)
                .orElseThrow(() -> new ProductNotFoundException(id));

        productMapper.updateEntity(existingProduct, request);
        Product updatedProduct = productRepository.save(existingProduct);

        return productMapper.toResponse(updatedProduct);
    }

    public void deleteById(Integer id) {
```

```
        if (!productRepository.existsById(id)) {
            throw new ProductNotFoundException(id);
        }

        productRepository.deleteById(id);
    }
}
```

12.1. Ce s-a îmbunătățit

Acum service-ul:

- nu mai întoarce `null` pentru cazuri problematice;
- folosește excepții specifice;
- lucrează cu mapper-ul;
- întoarce direct DTO-uri de răspuns;
- nu expune entitatea către exterior.

Această variantă este mai coerentă și reduce ramurile condiționale repetitive din controller.

13. Controller-ul devine mai zvelt (thin)

Fișier: `controller/ProductController.java`

```
package ro.unitbv.restlab.controller;

import jakarta.validation.Valid;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.service.ProductService;

import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public List<ProductResponse> getAllProducts() {
        return productService.findAll();
    }
```

```
    @GetMapping("/{id}")
    public ProductResponse getProductById(@PathVariable Integer id) {
        return productService.findById(id);
    }

    @PostMapping
    public ResponseEntity<ProductResponse> createProduct(
                        @Valid @RequestBody CreateProductRequest request) {
        ProductResponse createdProduct = productService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(createdProduct);
    }

    @PutMapping("/{id}")
    public ProductResponse updateProduct(
                        @PathVariable Integer id,
                        @Valid @RequestBody UpdateProductRequest request) {
        return productService.update(id, request);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        productService.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

13.1. Ce face @Valid

Adnotarea `@Valid` spune lui Spring că obiectul primit în request trebuie validat înainte ca metoda să continue. [2, 20]

De exemplu:

```
@Valid @RequestBody CreateProductRequest request
```

înseamnă că:

1. JSON-ul este deserializat într-un obiect `CreateProductRequest`;
2. se rulează regulile Bean Validation;
3. dacă există erori, metoda controller-ului nici măcar nu mai este executată;
4. excepție de validare este ridicată automat.

Acesta este un mecanism foarte elegant, deoarece mută validarea declarativă în DTO și evită verificările manuale repetitive.

14. Tratarea centralizată a erorilor

În acest punct apare una dintre cele mai importante îmbunătățiri arhitecturale: în loc să tratăm erorile separat în fiecare metodă, le centralizăm într-o clasă dedicată. [2]

Fișier: `exception/GlobalExceptionHandler.java`

```
package ro.unitbv.restlab.exception;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import ro.unitbv.restlab.dto.response.ErrorResponse;

import java.time.LocalDateTime;
import java.util.LinkedHashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(
                                            ProductNotFoundException ex,
                                            HttpServletRequest request) {
        ErrorResponse response = new ErrorResponse(
                LocalDateTime.now(),
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
                ex.getMessage(),
                request.getRequestURI() );

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationExceptionResponse>
           handleValidationException(MethodArgumentNotValidException ex,
                                     HttpServletRequest request) {

        Map<String, String> errors = new LinkedHashMap<>();

        ex.getBindingResult().getFieldErrors()
                             .forEach(fieldError ->
                                 errors.put(fieldError.getField(),
                                            fieldError.getDefaultMessage())
        );

        ValidationExceptionResponse response =
                                    new ValidationExceptionResponse(
                                           LocalDateTime.now(),
                                           HttpStatus.BAD_REQUEST.value(),
                                          "Validation failed",
                                          errors,
                                          request.getRequestURI()
                                       );
```

```
            return ResponseEntity.badRequest().body(response);
        }

        @ExceptionHandler(Exception.class)
        public ResponseEntity<ErrorResponse> handleGenericException(
                                                Exception ex,
                                                HttpServletRequest request) {
            ErrorResponse response = new ErrorResponse(
                    LocalDateTime.now(),
                    HttpStatus.INTERNAL_SERVER_ERROR.value(),
                    HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase(),
                    "A apărut o eroare internă neașteptată.",
                    request.getRequestURI()
            );

            return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                                 .body(response);
        }
    }
```

14.1. Ce face @RestControllerAdvice

`@RestControllerAdvice` definește o componentă globală care poate intercepta excepțiile apărute în controller-ele REST și poate construi răspunsuri HTTP uniforme. [2]

Practic, această clasă devine locul central în care se decide:

- ce cod HTTP se întoarce;
- ce structură JSON au erorile;
- cum sunt prezentate mesajele pentru client.

Avantajul este major: controller-ele rămân curate, iar răspunsurile de eroare devin consistente.

14.2. Ce face @ExceptionHandler

Fiecare metodă marcată cu `@ExceptionHandler(...)` spune cum trebuie tratat un anumit tip de excepție.

De exemplu:

```
@ExceptionHandler(ProductNotFoundException.class)
```

înseamnă că atunci când apare această excepție, Spring va folosi metoda respectivă pentru a construi răspunsul.

15. Exemplu de răspuns pentru produs inexistent

Dacă se apelează:

```
GET /api/products/999
```

iar produsul nu există, răspunsul poate fi:

```
{
  "timestamp": "2026-03-16T14:30:00.123",
  "status": 404,
  "error": "Not Found",
  "message": "Produsul cu id-ul 999 nu a fost găsit.",
  "path": "/api/products/999"
}
```

Acesta este un răspuns mult mai clar și mai util decât un simplu `404` fără context.

16. Exemplu de răspuns pentru eroare de validare

Dacă se trimite:

```
POST /api/products
Content-Type: application/json
```

cu body:

```
{
  "name": "",
  "price": 0,
  "stock": -2
}
```

răspunsul poate fi:

```
{
  "timestamp": "2026-03-16T14:32:10.456",
  "status": 400,
  "error": "Validation failed",
  "validationErrors": {
    "name": "Numele produsului este obligatoriu.",
    "price": "Prețul trebuie să fie mai mare decât 0.",
    "stock": "Stocul nu poate fi negativ."
  },
  "path": "/api/products"
}
```

Acest format este foarte util atât pentru debugging, cât și pentru clienți frontend care vor să afișeze mesaje de validare pe câmpuri.

17. Configurații și observații utile

17.1. Dependențe minime relevante

În `pom.xml`, pentru această variantă, sunt importante:

```
<dependencies>

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

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>

    <dependency>
        <groupId>com.mysql</groupId>
        <artifactId>mysql-connector-j</artifactId>
        <scope>runtime</scope>
    </dependency>

</dependencies>
```

17.2. Configurație tipică application.properties

```
spring.datasource.url=jdbc:mysql://localhost:3306/rest_lab_db?useSSL=false&serverTimezone=UTC
spring.datasource.username=labuser
spring.datasource.password=labpass
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false
spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect
```

Această configurație rămâne compatibilă cu varianta anterioară.

18. Documentarea API-ului cu OpenAPI și Swagger UI

Pe măsură ce API-ul devine mai bine structurat, apare o nevoie practică suplimentară: descrierea clară a contractului său. Un consumator al API-ului trebuie să poată afla ce endpoint-uri există, ce metode HTTP acceptă, ce parametri și DTO-uri sunt necesare și ce răspunsuri pot fi obținute. [4]

În ecosistemul Spring Boot, această problemă poate fi rezolvată folosind OpenAPI împreună cu Swagger UI. Cele două noțiuni nu sunt identice:

- OpenAPI este specificația care descrie contractul unui API într-o formă standardizată, independentă de interfața grafică;

- Swagger UI este o interfață web care citește descrierea OpenAPI și permite explorarea și apelarea interactivă a endpoint-urilor;
- springdoc-openapi integrează aceste mecanisme cu Spring Boot și generează descrierea OpenAPI pornind de la controller-ele, DTO-urile și adnotările aplicației.

Pentru proiectele Spring Boot 3.x folosite în acest îndrumar se utilizează linia 2.x a bibliotecii springdoc-openapi. O dependență Maven potrivită este:

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.9.1</version>
</dependency>
```

După adăugarea dependenței și pornirea aplicației nu este necesară o configurație suplimentară pentru cazul uzual Spring MVC. Descrierea și interfața interactivă sunt disponibile implicit la:

- `/v3/api-docs` – descrierea OpenAPI în format JSON;
- `/v3/api-docs.yaml` – aceeași descriere în format YAML;
- `/swagger-ui.html` – interfața Swagger UI.

Pentru aplicația locală din laborator, interfața poate fi deschisă în browser la:

```
http://localhost:8080/swagger-ui.html
```

18.1. Ce poate deduce automat springdoc-openapi

Fără adnotări suplimentare, biblioteca poate identifica o mare parte din contractul API pe baza structurii deja existente:

- căile definite prin `@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PutMapping` și `@DeleteMapping`;
- parametrii din URL și query string;
- corpurile de request descrise prin DTO-uri;
- tipurile de răspuns returnate de controller;
- o parte dintre constrângerile Bean Validation, precum `@NotBlank`, `@Size`, `@Min` sau `@DecimalMin`.

Acesta este un avantaj important al separării realizate în acest laborator: DTO-urile dedicate pentru request și response produc și un contract OpenAPI mai clar decât expunerea directă a entităților JPA. [4]

18.2. Îmbogățirea documentației cu adnotări OpenAPI

Generarea automată este suficientă pentru a obține rapid o documentație funcțională, dar numele și explicațiile pot fi îmbunătățite prin adnotările din pachetul `io.swagger.v3.oas.annotations`. De exemplu:

```
import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.tags.Tag;

@Tag(name = "Products", description = "Operații pentru catalogul de
produse")
@RestController
@RequestMapping("/api/products")
@RequiredArgsConstructor
public class ProductController {

    private final ProductService productService;

    @Operation(summary = "Listează toate produsele")
    @GetMapping
    public List<ProductResponse> getAllProducts() {
        return productService.findAll();
    }

    @Operation(summary = "Creează un produs nou")
    @PostMapping
    public ResponseEntity<ProductResponse> createProduct(
            @Valid @RequestBody CreateProductRequest request) {
        ProductResponse createdProduct = productService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(createdProduct);
    }
}
```

Adnotarea `@Tag` grupează logic operațiile, iar `@Operation` oferă o descriere lizibilă pentru fiecare endpoint. Pentru contracte mai detaliate pot fi folosite și `@ApiResponse`, `@Parameter` sau `@Schema`. [4]

De exemplu, un câmp dintr-un DTO poate fi explicat explicit astfel:

```
@Schema(description = "Numele afișat al produsului", example = "Laptop")
@NotBlank(message = "Numele produsului este obligatoriu.")
private String name;
```

18.3. Configurări utile

Căile implicite pot fi schimbate din configurația Spring Boot. De exemplu:

```
springdoc.swagger-ui.path=/docs
springdoc.api-docs.path=/api-docs
```

În acest caz, interfața Swagger UI va fi disponibilă la `/docs`, iar documentul OpenAPI JSON la `/api-docs`.

> **Notă privind securitatea**
>
> În laboratorul despre Spring Security, endpoint-urile de documentație pot deveni la rândul lor protejate. Dacă se dorește acces public la documentație într-un mediu de dezvoltare, regulile de securitate trebuie să permită explicit căile Swagger UI și OpenAPI. În producție, decizia trebuie luată în funcție de politica aplicației; documentația interactivă nu trebuie expusă automat doar pentru că este utilă în dezvoltare.

18.4. Testarea interactivă a endpoint-urilor

Swagger UI nu este doar o pagină de documentație. Pentru fiecare operație poate afișa parametrii, schema request-ului și exemplele de răspuns, iar butonul *Try it* out permite trimiterea unei cereri direct din browser. Pentru aplicația de laborator, aceasta oferă o alternativă rapidă la comenzi curl sau la un client REST separat atunci când se verifică manual endpoint-urile. [4]

Bună practică

OpenAPI trebuie tratat ca o descriere a contractului public al aplicației, nu ca un substitut pentru proiectarea API-ului. Este preferabil ca structura endpoint-urilor și DTO-urilor să fie clară prin ea însăși, iar adnotările Swagger/OpenAPI să adauge explicații utile, nu să repare un contract confuz.

19. Ce s-a schimbat conceptual în această elevare

Acum aplicația are un flux mai matur:

1. clientul trimite un request JSON;
2. JSON-ul este convertit într-un DTO de request;
3. DTO-ul este validat automat;
4. controller-ul delegă serviciului operația cerută;
5. service-ul folosește repository-ul și mapper-ul;
6. rezultatul este transformat într-un DTO de response;
7. dacă apare o problemă, excepția este tratată centralizat.

Acesta este deja un model foarte bun pentru aplicații REST de dimensiuni mici și medii.

20. Avantajele noii variante

În această formă, aplicația oferă mai multe beneficii clare:

- controller-ele sunt mai scurte și mai ușor de citit;
- contractul API este clar separat de modelul persistent;

- validările sunt declarative și apropiate de datele validate;
- erorile sunt uniforme și centralizate;
- service-ul exprimă mai bine logica aplicației;
- mapper-ul izolează conversiile între straturi;
- codul este mai ușor de extins și de testat.

Din punct de vedere didactic, aceasta este o formă foarte bună de „cod curat introductiv”, fără a introduce încă o complexitate prea mare.

21. Codul complet, în ordinea logică a rolurilor

21.1. Product.java

```
package ro.unitbv.restlab.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;

@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private double price;
    private int stock;

    public Product() {
    }

    public Product(Integer id, String name, double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Product(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public Integer getId() {
        return id;
    }
```

```
    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

21.2. CreateProductRequest.java

```
package ro.unitbv.restlab.dto.request;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class CreateProductRequest {

    @NotBlank(message = "Numele produsului este obligatoriu.")
    @Size(max = 100,
          message = "Numele produsului nu poate depăși 100 de caractere.")
    private String name;

    @DecimalMin(value = "0.01",
                message = "Prețul trebuie să fie mai mare decât 0.")
    private double price;

    @Min(value = 0, message = "Stocul nu poate fi negativ.")
    private int stock;

    public CreateProductRequest() { }
```

```
    public CreateProductRequest(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

21.3. UpdateProductRequest.java

```
package ro.unitbv.restlab.dto.request;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public class UpdateProductRequest {

    @NotBlank(message = "Numele produsului este obligatoriu.")
    @Size(max = 100, message = "Numele produsului nu poate depăși 100 de
caractere.")
    private String name;

    @DecimalMin(value = "0.01", message = "Prețul trebuie să fie mai mare
decât 0.")
    private double price;

    @Min(value = 0, message = "Stocul nu poate fi negativ.")
    private int stock;
    public UpdateProductRequest() {
    }
```

```
    public UpdateProductRequest(String name, double price, int stock) {
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

21.4. ProductResponse.java

```
package ro.unitbv.restlab.dto.response;

public class ProductResponse {

    private Integer id;
    private String name;
    private double price;
    private int stock;

    public ProductResponse() {
    }

    public ProductResponse(Integer id, String name,
                           double price, int stock) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }
```

```
    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getPrice() {
        return price;
    }

    public void setPrice(double price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

21.5. ErrorResponse.java

```
package ro.unitbv.restlab.dto.response;

import java.time.LocalDateTime;
public class ErrorResponse {

    private LocalDateTime timestamp;
    private int status;
    private String error;
    private String message;
    private String path;

    public ErrorResponse() { }

    public ErrorResponse(LocalDateTime timestamp, int status,
                         String error, String message, String path) {
        this.timestamp = timestamp;
        this.status = status;
        this.error = error;
        this.message = message;
```

```
        this.path = path;
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }

    public String getError() {
        return error;
    }

    public void setError(String error) {
        this.error = error;
    }

    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    public String getPath() {
        return path;
    }

    public void setPath(String path) {
        this.path = path;
    }
}
```

21.6. ValidationExceptionResponse.java

```
package ro.unitbv.restlab.exception;

import java.time.LocalDateTime;
import java.util.Map;

public class ValidationExceptionResponse {

    private LocalDateTime timestamp;
    private int status;
    private String error;
    private Map<String, String> validationErrors;
```

```
    private String path;

    public ValidationExceptionResponse() {
    }

    public ValidationExceptionResponse(
                                LocalDateTime timestamp,
                                int status,
                                String error,
                                Map<String, String> validationErrors,
                                String path) {
        this.timestamp = timestamp;
        this.status = status;
        this.error = error;
        this.validationErrors = validationErrors;
        this.path = path;
    }

    public LocalDateTime getTimestamp() {
        return timestamp;
    }
    public void setTimestamp(LocalDateTime timestamp) {
        this.timestamp = timestamp;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }

    public String getError() {
        return error;
    }

    public void setError(String error) {
        this.error = error;
    }

    public Map<String, String> getValidationErrors() {
        return validationErrors;
    }

    public void setValidationErrors(Map<String, String> validationErrors)
    {
        this.validationErrors = validationErrors;
    }

    public String getPath() {
        return path;
    }
```

```
    public void setPath(String path) {
        this.path = path;
    }
}
```

21.7. ProductNotFoundException.java

```
package ro.unitbv.restlab.exception;

public class ProductNotFoundException extends RuntimeException {

    public ProductNotFoundException(Integer id) {
        super("Produsul cu id-ul " + id + " nu a fost găsit.");
    }
}
```

21.8. ProductMapper.java

```
package ro.unitbv.restlab.mapper;

import org.springframework.stereotype.Component;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.model.Product;

@Component
public class ProductMapper {

    public Product toEntity(CreateProductRequest request) {
        return new Product(
                request.getName(),
                request.getPrice(),
                request.getStock()
        );
    }

    public void updateEntity(Product product,
                             UpdateProductRequest request) {
        product.setName(request.getName());
        product.setPrice(request.getPrice());
        product.setStock(request.getStock());
    }

    public ProductResponse toResponse(Product product) {
        return new ProductResponse(
                product.getId(),
                product.getName(),
                product.getPrice(),
                product.getStock()
        );
    }
}
```

21.9. ProductRepository.java

```
package ro.unitbv.restlab.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import ro.unitbv.restlab.model.Product;

public interface ProductRepository extends JpaRepository<Product, Integer>
{
}
```

21.10. ProductService.java

```
package ro.unitbv.restlab.service;

import org.springframework.stereotype.Service;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.mapper.ProductMapper;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;

import java.util.List;

@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    public ProductService(ProductRepository productRepository,
                          ProductMapper productMapper) {
        this.productRepository = productRepository;
        this.productMapper = productMapper;
    }

    public List<ProductResponse> findAll() {
        return productRepository.findAll()
                .stream()
                .map(productMapper::toResponse)
                .toList();
    }

    public ProductResponse findById(Integer id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ProductNotFoundException(id));

        return productMapper.toResponse(product);
    }

    public ProductResponse create(CreateProductRequest request) {
        Product product = productMapper.toEntity(request);
```

```
        Product savedProduct = productRepository.save(product);
        return productMapper.toResponse(savedProduct);
    }

    public ProductResponse update(Integer id,
                                  UpdateProductRequest request) {
        Product existingProduct = productRepository.findById(id)
                .orElseThrow(() -> new ProductNotFoundException(id));

        productMapper.updateEntity(existingProduct, request);
        Product updatedProduct = productRepository.save(existingProduct);

        return productMapper.toResponse(updatedProduct);
    }

    public void deleteById(Integer id) {
        if (!productRepository.existsById(id)) {
            throw new ProductNotFoundException(id);
        }

        productRepository.deleteById(id);
    }
}
```

21.11. ProductController.java

```
package ro.unitbv.restlab.controller;

import jakarta.validation.Valid;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.service.ProductService;

import java.util.List;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }

    @GetMapping
    public List<ProductResponse> getAllProducts() {
        return productService.findAll();
    }
```

```
    @GetMapping("/{id}")
    public ProductResponse getProductById(@PathVariable Integer id) {
        return productService.findById(id);
    }

    @PostMapping
    public ResponseEntity<ProductResponse> createProduct(
                    @Valid @RequestBody CreateProductRequest request) {
        ProductResponse createdProduct = productService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(createdProduct);
    }

    @PutMapping("/{id}")
    public ProductResponse updateProduct(
                    @PathVariable Integer id,
                    @Valid @RequestBody UpdateProductRequest request) {
        return productService.update(id, request);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        productService.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

21.12. GlobalExceptionHandler.java

```
package ro.unitbv.restlab.exception;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;

import ro.unitbv.restlab.dto.response.ErrorResponse;

import java.time.LocalDateTime;
import java.util.LinkedHashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(
                                            ProductNotFoundException ex,
                                            HttpServletRequest request) {
        ErrorResponse response = new ErrorResponse(
                LocalDateTime.now(),
                HttpStatus.NOT_FOUND.value(),
                HttpStatus.NOT_FOUND.getReasonPhrase(),
```

```
                ex.getMessage(),
                request.getRequestURI()
        );

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationExceptionResponse>
handleValidationException(
            MethodArgumentNotValidException ex,
            HttpServletRequest request) {

        Map<String, String> errors = new LinkedHashMap<>();

        ex.getBindingResult().getFieldErrors()
                             .forEach(fieldError ->
                                  errors.put(fieldError.getField(),
                                  fieldError.getDefaultMessage())
                             );
        ValidationExceptionResponse response =
            new ValidationExceptionResponse(
                   LocalDateTime.now(),
                   HttpStatus.BAD_REQUEST.value(),
                   "Validation failed",
                   errors,
                   request.getRequestURI()
            );

        return ResponseEntity.badRequest().body(response);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(
                                            Exception ex,
                                            HttpServletRequest request) {
        ErrorResponse response = new ErrorResponse(
                LocalDateTime.now(),
                HttpStatus.INTERNAL_SERVER_ERROR.value(),
                HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase(),
                "A apărut o eroare internă neașteptată.",
                request.getRequestURI()
        );

        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body(response);
    }
}
```

22. Utilizarea Lombok în aplicații Spring Boot

Project Lombok este o bibliotecă Java care reduce semnificativ boilerplate-ul prin generarea automată, la compilare, a unor elemente repetitive precum:

- getter/setter
- constructori
- metode equals(), hashCode(), toString()
- builderi pentru obiecte complexe

În contextul aplicațiilor Spring Boot, unde modelarea DTO-urilor, entităților JPA și claselor de configurare implică multe clase de tip container de date (*data holder*), Lombok aduce beneficii clare.

22.1. Reducerea codului repetitiv

Clasele devin mult mai concise. De exemplu:

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class Product {
    private Long id;
    private String name;
    private double price;
}
```

înlocuiește zeci de linii de cod scrise manual (sau generate cu ajutorul IDE-ului).

22.2. Creșterea lizibilității

Se evidențiază mai bine structura logică a clasei (ce date conține), nu detaliile tehnice repetitive.

22.3. Productivitate crescută

Dezvoltatorii pot scrie mai rapid cod și se concentrează pe logică de business, nu pe infrastructură trivială.

22.4. Integrare bună cu Spring Boot

Lombok este frecvent utilizat împreună cu:

- DTO-uri (`@Data`, `@Builder`)
- entități JPA (`@Getter`, `@Setter`)
- clase de configurare (`@RequiredArgsConstructor` pentru dependency injection)

22.5. Avantaje (când este recomandat)

- proiecte enterprise cu multe DTO-uri și entități
- echipe care acceptă dependențe externe și au tooling configurat corect

- aplicații Spring Boot moderne (unde Lombok este aproape standard de facto)

22.6. Dezavantaje / riscuri

- Magie la compilare (annotation processing): codul generat nu este vizibil direct → poate crea dificultăți pentru începători.
- Dependență externă: dacă Lombok dispare sau nu mai este întreținut, migrarea poate fi costisitoare (deși riscul este mic în prezent).
- Debugging mai dificil: stacktrace-ul include cod generat automat.
- Probleme cu unele tool-uri: IDE-ul trebuie configurat corect (annotation processing activ).

22.7. Recomandări de bună practică

În aplicații Spring Boot, utilizarea Lombok este indicată dacă respecți câteva reguli:

- **Nu se folosește `@Data` pe entități JPA** → se preferă `@Getter` + `@Setter` (pentru control mai bun)
- **Utilizarea `@Builder` pentru DTO-uri** → mai ales în request/response complexe
- **Folosește `@RequiredArgsConstructor` pentru injection**

```
@Service
@RequiredArgsConstructor
public class ProductService {
    private final ProductRepository repository;
}
```

- **Totuși e bine să se evite abuzul de adnotări** → pentru a păstra claritatea codului

22.8. Dependența necesară în pom.xml

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

Dacă proiectul este compilat cu Maven, este recomandat ca Lombok să fie inclus și pe calea de annotation processing a compilatorului, pentru ca generarea codului să fie robustă și predictibilă în toate mediile.

22.9. Adnotări utile de utilizat în aplicații Spring Boot

Adnotare	*Ce generează*	*Utilizare tipică*	*Observații*
`@Getter`	Metode `getX()` pentru toate câmpurile (sau selectiv)	DTO, entități JPA	Poate fi pusă și pe clasă sau pe câmp
`@Setter`	Metode `setX()` pentru câmpuri	DTO, entități	Evită pe câmpuri final
`@ToString`	Metoda `toString()`	Debugging	Atenție la entități JPA (lazy loading)
`@EqualsAndHashCode`	`equals()` + `hashCode()`	DTO-uri	**Evită pe entități JPA fără configurare**

`@NoArgsConstructor`	Constructor fără argumente	Entități JPA (obligatoriu)	Poate fi `force = true`
`@AllArgsConstructor`	Constructor cu toate câmpurile	DTO-uri	Nu e ideal pentru entități
`@RequiredArgsConstructor`	Constructor cu câmpuri final sau `@NonNull`	Dependency Injection (Spring)	**Foarte recomandat**
`@Data`	`@Getter + @Setter + @ToString + @EqualsAndHashCode + @RequiredArgsConstructor`	DTO-uri simple	**Evită pe entități JPA**
`@Builder`	Pattern Builder	DTO-uri complexe	**Foarte util pentru request/response**
`@Value`	Clasă imutabilă (finală, fără setteri)	DTO-uri read-only	Alternativă la `@Data`
`@NonNull`	Validare null + check în constructor	Parametri critici	Aruncă `NullPointerException`
`@Slf4j`	Logger private static final Logger log	Logging	Evită declararea manuală
`@SneakyThrows`	Elimină necesitatea throws/try-catch	Cod tehnic	Folosește cu grijă
`@Builder.Default`	Valoare implicită în builder	DTO-uri cu valori default	Necesită `@Builder`
`@Accessors(chain = true)`	Setteri chainable (`obj.setX().setY()`)	Fluent API	Opțional
`@FieldDefaults`	Setează nivel de acces implicit	Reducere boilerplate	Mai rar folosit

22.10. Dezvoltarea aplicației folosind Lombok

În continuare se prezintă o versiune completă a aplicației dezvoltate, adaptată pentru a utiliza biblioteca Lombok. Scopul acestei rescrieri este de a evidenția modul în care adnotările Lombok simplifică semnificativ codul, fără a afecta structura arhitecturală a aplicației.

Implementarea păstrează aceleași straturi logice:

- entități JPA pentru modelul de date,
- DTO-uri pentru transferul datelor,
- mapper pentru conversia între modele,
- service pentru logica de business,
- controller pentru expunerea API-ului REST.

Diferența esențială constă în eliminarea codului repetitiv (getteri, setteri, constructori etc.), acesta fiind generat automat prin mecanismul de *annotation processing* oferit de Lombok.

Prin această abordare se obțin:

- reducere semnificativă a dimensiunii codului,
- lizibilitate mai bună a claselor,
- separare mai clară între structura datelor și logica aplicației.

Codul generat de Lombok este adăugat în faza de compilare, fiind complet transparent pentru JVM în timpul execuției.

În cadrul acestui exemplu, aplicația va utiliza baza de date H2, o bază de date relațională in-memory, frecvent folosită în aplicații Java pentru dezvoltare și testare. H2 rulează direct în memoria aplicației (sau opțional pe disc), nu necesită instalare externă și pornește automat împreună cu aplicația Spring Boot, ceea ce simplifică semnificativ configurarea mediului de lucru. Datorită suportului pentru SQL standard și integrării native cu Spring Boot, H2 permite testarea rapidă a funcționalităților fără dependențe externe, fiind ideală pentru aplicații didactice sau prototipuri. În acest context, schema bazei de date este generată automat de JPA/Hibernate.

22.10.1. pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.3.5</version>
        <relativePath/>
    </parent>
    <groupId>ro.unitbv</groupId>
    <artifactId>restlab-l6-lombok</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>restlab-l6-lombok</name>

    <properties>
        <java.version>21</java.version>
    </properties>

    <dependencies>

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

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

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

```
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <annotationProcessorPaths>
                        <path>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </path>
                    </annotationProcessorPaths>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

22.10.2. src/main/java/ro/unitbv/restlab/RestLabApplication.java

```
package ro.unitbv.restlab;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class RestLabApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(RestLabApplication.class, args);
    }
}
```

22.10.3. src/main/java/ro/unitbv/restlab/model/Product.java

```
package ro.unitbv.restlab.model;

import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    private String name;
    private double price;
    private int stock;
}
```

22.10.4. src/main/java/ro/unitbv/restlab/dto/request/CreateProductRequest.java

```
package ro.unitbv.restlab.dto.request;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
```

```
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class CreateProductRequest {

    @NotBlank(message = "Numele produsului este obligatoriu.")
    @Size(max = 100,
          message = "Numele produsului nu poate depasi 100 de caractere.")
    private String name;

    @DecimalMin(value = "0.01",
                message = "Pretul trebuie sa fie mai mare decat 0.")
    private double price;

    @Min(value = 0, message = "Stocul nu poate fi negativ.")
    private int stock;
}
```

22.10.5. src/main/java/ro/unitbv/restlab/dto/request/UpdateProductRequest.java

```
package ro.unitbv.restlab.dto.request;

import jakarta.validation.constraints.DecimalMin;
import jakarta.validation.constraints.Min;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class UpdateProductRequest {

    @NotBlank(message = "Numele produsului este obligatoriu.")
    @Size(max = 100,
          message = "Numele produsului nu poate depasi 100 de caractere.")
    private String name;

    @DecimalMin(value = "0.01",
                message = "Pretul trebuie sa fie mai mare decat 0.")
    private double price;

    @Min(value = 0, message = "Stocul nu poate fi negativ.")
    private int stock;
}
```

22.10.6. src/main/java/ro/unitbv/restlab/dto/response/ProductResponse.java

```
package ro.unitbv.restlab.dto.response;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ProductResponse {

    private Integer id;
    private String name;
    private double price;
    private int stock;
}
```

22.10.7. src/main/java/ro/unitbv/restlab/dto/response/ErrorResponse.java

```
package ro.unitbv.restlab.dto.response;

import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.LocalDateTime;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ErrorResponse {

    private LocalDateTime timestamp;
    private int status;
    private String error;
    private String message;
    private String path;
}
```

22.10.8. src/main/java/ro/unitbv/restlab/dto/response/ValidationExceptionResponse.java

```
package ro.unitbv.restlab.dto.response;
```

```
import lombok.AllArgsConstructor;
import lombok.Builder;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.LocalDateTime;
import java.util.Map;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class ValidationExceptionResponse {

    private LocalDateTime timestamp;
    private int status;
    private String error;
    private Map<String, String> validationErrors;
    private String path;
}
```

22.10.9. src/main/java/ro/unitbv/restlab/exception/ProductNotFoundException.java

```
package ro.unitbv.restlab.exception;

public class ProductNotFoundException extends RuntimeException {

    public ProductNotFoundException(Integer id) {
        super("Produsul cu id-ul " + id + " nu a fost gasit.");
    }
}
```

22.10.10. src/main/java/ro/unitbv/restlab/mapper/ProductMapper.java

```
package ro.unitbv.restlab.mapper;

import org.springframework.stereotype.Component;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.model.Product;

@Component
public class ProductMapper {

    public Product toEntity(CreateProductRequest request) {
        return Product.builder()
                .name(request.getName())
                .price(request.getPrice())
```

```
                .stock(request.getStock())
                .build();
    }

    public void updateEntity(Product product,
                             UpdateProductRequest request) {
        product.setName(request.getName());
        product.setPrice(request.getPrice());
        product.setStock(request.getStock());
    }

    public ProductResponse toResponse(Product product) {
        return ProductResponse.builder()
                .id(product.getId())
                .name(product.getName())
                .price(product.getPrice())
                .stock(product.getStock())
                .build();
    }
}
```

22.10.11. src/main/java/ro/unitbv/restlab/repository/ProductRepository.java

```
package ro.unitbv.restlab.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import ro.unitbv.restlab.model.Product;

public interface ProductRepository extends JpaRepository<Product, Integer>
{  }
```

22.10.12. src/main/java/ro/unitbv/restlab/service/ProductService.java

```
package ro.unitbv.restlab.service;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.mapper.ProductMapper;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;

import java.util.List;

@Service
@RequiredArgsConstructor
public class ProductService {

    private final ProductRepository productRepository;
```

```
    private final ProductMapper productMapper;

    public List<ProductResponse> findAll() {
        return productRepository.findAll()
                .stream()
                .map(productMapper::toResponse)
                .toList();
    }

    public ProductResponse findById(Integer id) {
        Product product = productRepository.findById(id)
                .orElseThrow(() -> new ProductNotFoundException(id));

        return productMapper.toResponse(product);
    }

    public ProductResponse create(CreateProductRequest request) {
        Product product = productMapper.toEntity(request);
        Product savedProduct = productRepository.save(product);
        return productMapper.toResponse(savedProduct);
    }

    public ProductResponse update(Integer id,
                                  UpdateProductRequest request) {
        Product existingProduct = productRepository.findById(id)
                .orElseThrow(() -> new ProductNotFoundException(id));

        productMapper.updateEntity(existingProduct, request);
        Product updatedProduct = productRepository.save(existingProduct);

        return productMapper.toResponse(updatedProduct);
    }

    public void deleteById(Integer id) {
        if (!productRepository.existsById(id)) {
            throw new ProductNotFoundException(id);
        }

        productRepository.deleteById(id);
    }
}
```

22.10.13. src/main/java/ro/unitbv/restlab/controller/ProductController.java

```
package ro.unitbv.restlab.controller;

import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
```

```
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.service.ProductService;
import java.util.List;

@RestController
@RequestMapping("/api/products")
@RequiredArgsConstructor
public class ProductController {

    private final ProductService productService;

    @GetMapping
    public List<ProductResponse> getAllProducts() {
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public ProductResponse getProductById(@PathVariable Integer id) {
        return productService.findById(id);
    }

    @PostMapping
    public ResponseEntity<ProductResponse> createProduct(
                    @Valid @RequestBody CreateProductRequest request) {
        ProductResponse createdProduct = productService.create(request);

        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(createdProduct);
    }

    @PutMapping("/{id}")
    public ProductResponse updateProduct(
                      @PathVariable Integer id,
                      @Valid @RequestBody UpdateProductRequest request) {
        return productService.update(id, request);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        productService.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

22.10.14. src/main/java/ro/unitbv/restlab/exception/GlobalExceptionHandler.java

```
package ro.unitbv.restlab.exception;

import jakarta.servlet.http.HttpServletRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
```

```
import org.springframework.web.bind.annotation.RestControllerAdvice;
import ro.unitbv.restlab.dto.response.ErrorResponse;
import ro.unitbv.restlab.dto.response.ValidationExceptionResponse;

import java.time.LocalDateTime;
import java.util.LinkedHashMap;
import java.util.Map;

@RestControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(
                                            ProductNotFoundException ex,
                                            HttpServletRequest request) {
        ErrorResponse response = ErrorResponse.builder()
                .timestamp(LocalDateTime.now())
                .status(HttpStatus.NOT_FOUND.value())
                .error(HttpStatus.NOT_FOUND.getReasonPhrase())
                .message(ex.getMessage())
                .path(request.getRequestURI())
                .build();

        return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationExceptionResponse>
           handleValidationException(
                           MethodArgumentNotValidException ex,
                           HttpServletRequest request) {

        Map<String, String> validationErrors = new LinkedHashMap<>();

        ex.getBindingResult().getFieldErrors()
                             .forEach(error ->
                validationErrors.put(error.getField(),
                                     error.getDefaultMessage()));

        ValidationExceptionResponse response = ValidationExceptionResponse
                .builder()
                .timestamp(LocalDateTime.now())
                .status(HttpStatus.BAD_REQUEST.value())
                .error(HttpStatus.BAD_REQUEST.getReasonPhrase())
                .validationErrors(validationErrors)
                .path(request.getRequestURI())
                .build();

        return ResponseEntity.badRequest().body(response);
    }

    @ExceptionHandler(Exception.class)
    public ResponseEntity<ErrorResponse> handleGenericException(
                                            Exception ex,
```

```
                                                HttpServletRequest request) {
        ErrorResponse response = ErrorResponse.builder()
                .timestamp(LocalDateTime.now())
                .status(HttpStatus.INTERNAL_SERVER_ERROR.value())
                .error(HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase())
                .message("A aparut o eroare interna.")
                .path(request.getRequestURI())
                .build();

        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
                             .body(response);
    }
}
```

22.10.15. src/main/resources/application.properties

```
spring.application.name=restlab-l6-lombok

spring.datasource.url=jdbc:h2:mem:restlab;MODE=MySQL;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

23. Întrebări recapitulative

1. De ce este util să existe DTO-uri separate pentru request și response?
2. Ce problemă rezolvă `@Valid`?
3. Unde este cel mai potrivit loc pentru logica de conversie dintre entitate și DTO?
4. De ce este mai curat să aruncăm o excepție specifică decât să întoarcem `null`?
5. Ce rol are `@RestControllerAdvice`?
6. Care este avantajul unei structuri uniforme pentru răspunsurile de eroare?
7. De ce este util ca service-ul să întoarcă DTO-uri de response în această variantă?
8. Ce câștigăm prin faptul că entitatea `Product` nu mai este expusă direct prin API?
9. Care este diferența dintre specificația OpenAPI, Swagger UI și biblioteca springdoc-openapi?
10. Ce informații despre un endpoint poate deduce automat springdoc-openapi și când sunt utile adnotările precum `@Operation` sau `@Schema`?

24. Exerciții propuse

24.1. Exercițiul 1

Adăugați câmpul `category` pentru produs și actualizați:

- entitatea;
- DTO-urile de request;
- DTO-ul de response;
- mapper-ul;
- validările.

24.2. Exercițiul 2

Adăugați în `ProductRepository` o metodă de căutare după nume și expuneți un endpoint:

```
GET /api/products/search?name=lap
```

care returnează `List<ProductResponse>`.

24.3. Exercițiul 3

Introduceți și un DTO separat pentru actualizarea parțială a stocului, de exemplu `UpdateStockRequest`.

24.4. Exercițiul 4

Extindeți `GlobalExceptionHandler` astfel încât să trateze și situațiile în care JSON-ul trimis de client este invalid sintactic.

24.5. Exercițiul 5

Adăugați un câmp `createdAt` în entitate și decideți dacă trebuie sau nu expus în `ProductResponse`.

24.6. Exercițiul 6

Adăugați dependența *springdoc-openapi-starter-webmvc-ui*, porniți aplicația și verificați documentația generată la `/swagger-ui.html`. Adăugați apoi `@Tag` pe controller și cel puțin două adnotări `@Operation` pe endpoint-uri. Comparați documentul OpenAPI de la `/v3/api-docs` înainte și după adăugarea adnotărilor.

Laborator 4: De la funcționare la monitorizare – logging și metrici

1. Logging în aplicații Spring Boot

În etapele anterioare, aplicația REST a fost adusă într-o formă mai matură: datele de intrare și de ieșire sunt separate prin DTO-uri, validarea este realizată automat prin Bean Validation, erorile sunt tratate centralizat prin `@RestControllerAdvice`, iar biblioteca Lombok reduce o parte importantă din codul repetitiv. În această fază, aplicația nu mai este doar „funcțională", ci are deja o structură suficient de clară pentru a putea fi observată și analizată în execuție. [1, 23]

O aplicație reală nu trebuie doar să răspundă corect la cereri. Ea trebuie să ofere și informații despre ceea ce face în timpul rulării. În practică, apar foarte repede întrebări precum: ce cerere a ajuns în aplicație, ce metodă a fost apelată, cu ce identificator s-a lucrat, de ce a eșuat o operație sau în ce punct exact s-a produs o excepție. Fără un mecanism de logging, răspunsul la astfel de întrebări devine dificil, iar depanarea se transformă într-un proces lent, bazat pe presupuneri.

Logging-ul reprezintă mecanismul prin care aplicația produce mesaje controlate despre propria execuție. Aceste mesaje pot descrie fluxul normal al unei operații, situații neobișnuite, avertismente sau erori. În mod ideal, logurile trebuie să ajute dezvoltatorul sau administratorul să înțeleagă ce face aplicația, fără a aglomera inutil consola sau fișierele de jurnalizare.

În contextul aplicației REST dezvoltate anterior, logging-ul este deosebit de util deoarece arhitectura este deja suficient de clară: controller-ul primește cereri HTTP, service-ul conține logica aplicației, repository-ul realizează accesul la date, iar handler-ul global de excepții centralizează erorile. Această separare face posibilă introducerea logurilor exact în punctele potrivite.

1.1. De ce este necesar logging-ul

Necesitatea logging-ului apare imediat ce aplicația începe să fie folosită în mod real. În timpul dezvoltării, programatorul are nevoie să observe traseul unei cereri și să înțeleagă dacă datele primite au produs efectul așteptat. În timpul testării, logurile ajută la verificarea faptului că aplicația parcurge fluxul corect și că erorile sunt gestionate în mod controlat. În exploatare, ele oferă informații esențiale atunci când ceva nu funcționează așa cum ar trebui.

Un prim rol important al logging-ului este **diagnosticarea erorilor**. Dacă un endpoint întoarce `404 Not Found`, `400 Bad Request` sau `500 Internal Server Error`, logurile pot

arăta de ce s-a produs acel răspuns. În aplicația dezvoltată anterior, de exemplu, dacă produsul cu un anumit ID nu există, se aruncă `ProductNotFoundException`, iar această situație poate fi înregistrată în log cu un mesaj clar. În mod similar, dacă validarea unui request eșuează, handler-ul global poate consemna că au existat erori de validare pentru un anumit endpoint.

Un al doilea rol este **înțelegerea fluxului de execuție**. În aplicațiile stratificate, request-ul nu este procesat într-un singur loc. El trece prin controller, service și, eventual, repository. Logging-ul permite urmărirea acestui traseu. De exemplu, se poate observa că un endpoint `GET /api/products/5` a fost apelat, că service-ul a început căutarea produsului cu ID 5 și că operația s-a finalizat cu succes sau cu o excepție.

Un al treilea rol este **observarea comportamentului aplicației în timp**. Prin loguri se poate vedea dacă există multe cereri invalide, dacă anumite operații sunt apelate foarte frecvent sau dacă apar repetat aceleași situații anormale.

Un al patrulea rol este **separarea dintre informația tehnică internă și răspunsul oferit clientului**. Clientul API-ului trebuie să primească un răspuns HTTP clar și curat, sub forma unui JSON bine structurat. În schimb, logurile pot conține informații suplimentare utile doar pentru dezvoltator: clasa în care s-a produs problema, nivelul logului, momentul exact, eventual chiar și stacktrace-ul unei excepții.

1.2. De ce System.out.println(...) nu este suficient

În programele foarte mici sau în exercițiile introductive, este tentant să se folosească instrucțiuni precum:

```
System.out.println("A intrat în metoda create()");
```

Această abordare poate părea comodă la început, însă nu este potrivită pentru aplicații Spring Boot serioase. Problema principală este că `System.out.println(...)` nu oferă niciun mecanism real de organizare și control al mesajelor produse.

În primul rând, mesajele afișate astfel nu au **niveluri de severitate**. Nu se poate distinge clar între un mesaj pur informativ, un mesaj util pentru debugging, un avertisment sau o eroare critică. În al doilea rând, nu există **configurare centralizată**: nu se poate spune ușor că vrem să vedem doar anumite mesaje din pachetul aplicației sau că vrem să ascundem logurile prea detaliate. În al treilea rând, ieșirea nu este **structurată**: de regulă, nu există un format standard cu timestamp, nivel, numele clasei și mesaj.

Framework-urile de logging rezolvă exact aceste probleme. Ele permit definirea unor niveluri de logare, configurarea lor per aplicație sau per pachet, formatarea uniformă a mesajelor și redirecționarea acestora către consolă, fișiere sau alte sisteme. Din acest motiv, în aplicațiile moderne nu se recomandă folosirea lui `System.out.println(...)` pentru observarea execuției.

1.3. Ce este un logger

Un **logger** este obiectul prin care o clasă produce mesaje de log. Prin intermediul lui se pot emite mesaje la diferite niveluri, cum ar fi `trace`, `debug`, `info`, `warn` și `error`.

În practică, aproape fiecare clasă importantă are propriul logger. Astfel, atunci când apare un mesaj în consolă sau într-un fișier de log, se poate identifica imediat clasa din care provine. Acest lucru este foarte util în aplicațiile stratificate, deoarece diferențiază clar mesajele venite din controller, din service sau din handler-ul global de excepții. [1]

În ecosistemul Spring Boot, loggerul este folosit de regulă prin intermediul fațadei[11] SLF4J[12], iar implementarea implicită este Logback. Din punctul de vedere al programatorului, ceea ce contează cel mai mult este că poate scrie cod standardizat de logging, iar Spring Boot se ocupă de integrarea cu infrastructura concretă.

1.4. Infrastructura de logging în Spring Boot

Spring Boot folosește implicit o combinație foarte cunoscută în ecosistemul Java: **SLF4J** și **Logback**[13].

1.4.1. SLF4J

SLF4J este o fațadă de logging. Asta înseamnă că el oferă un API comun prin care aplicația scrie loguri, fără a depinde direct de o implementare concretă. În cod, atunci când se declară un `Logger` și se folosește `LoggerFactory`, de fapt se lucrează cu API-ul SLF4J. [1]

Avantajul acestei abordări este că aplicația nu este legată direct de un singur framework de logging. Programatorul scrie cod împotriva unei interfețe standard, iar implementarea efectivă poate fi schimbată mai ușor.

1.4.2. Logback

Logback este implementarea concretă folosită implicit de Spring Boot pentru a scrie efectiv logurile. El se ocupă de:

- formatarea mesajelor;
- afișarea lor în consolă;
- scrierea lor în fișiere, dacă este configurat;
- filtrarea lor în funcție de nivel.

În multe proiecte Spring Boot standard, programatorul nu trebuie să configureze explicit Logback de la început. Framework-ul oferă o configurație implicită suficient de bună pentru

[11] o fațadă precum SLF4J este un API intermediar care oferă o interfață unificată de logare și decuplează aplicația de implementările concrete (ex. Logback, Log4j2), permițând schimbarea acestora fără modificări în cod.

[12] denumirea SLF4J provine de la „Simple Logging Facade for Java”, indicând faptul că este o fațadă simplă de logare pentru aplicații Java.

[13] Logback este un framework de logging pentru Java, implementare nativă pentru SLF4J, care gestionează efectiv scrierea și configurarea logurilor (appendere, formatare, rotație etc.).

dezvoltare, iar ulterior aceasta poate fi ajustată prin fișiere de proprietăți sau configurații dedicate.

1.5. Nivelurile de logare

Un aspect esențial în logging este existența **nivelurilor de logare**. Acestea permit clasificarea mesajelor în funcție de importanță și severitate. Ordinea uzuală este:

```
TRACE < DEBUG < INFO < WARN < ERROR
```

Această ordine este importantă deoarece filtrarea se face pe baza ei. Dacă nivelul configurat este `INFO`, atunci mesajele `INFO`, `WARN` și `ERROR` vor fi afișate, în timp ce mesajele `DEBUG` și `TRACE` vor fi ignorate.

Nivelurile fac posibil un control foarte util. În timpul dezvoltării, se poate activa `DEBUG` pentru a vedea mai multe detalii despre ceea ce face aplicația. În producție, de regulă, se folosesc mai ales `INFO`, `WARN` și `ERROR`, pentru a evita aglomerarea logurilor cu informații prea fine. [1]

1.5.1. Tabel comparativ al nivelurilor de log

Nivel	*Semnificație*	*Când se utilizează*	*Exemplu*
`TRACE`	detalii foarte fine ale execuției	urmărirea foarte detaliată a fluxului intern	pași foarte mici ai unui algoritm
`DEBUG`	informații utile pentru dezvoltare și depanare	când vrem să înțelegem deciziile interne ale codului	căutarea unui produs după ID
`INFO`	evenimente normale și importante	operații relevante executate cu succes	produs creat, request procesat
`WARN`	situații anormale, dar controlate	când ceva nu este ideal, dar aplicația continuă	produs inexistent, erori de validare
`ERROR`	erori reale care afectează operația	excepții și probleme serioase	eroare internă neașteptată

1.6. Diferența dintre nivelurile de log

1.6.1. TRACE

`TRACE` este cel mai detaliat nivel. El este folosit atunci când dorim să urmărim foarte fin execuția internă a aplicației. Într-o aplicație REST simplă, acest nivel este rareori necesar. Totuși, merită cunoscut, deoarece în aplicații mai complexe poate fi util pentru diagnosticarea unor fluxuri greu de urmărit sau a unor algoritmi sofisticați.

Un exemplu ar fi logarea fiecărui pas dintr-o prelucrare internă detaliată, lucru care, în proiectul cu produse, nu este în general necesar.

1.6.2. DEBUG

`DEBUG` este destinat mai ales dezvoltării și depanării. El oferă informații utile despre ce face aplicația, fără a intra în granularitatea extremă a nivelului `TRACE`. În practica de zi cu zi, `DEBUG` este foarte util pentru:

- identificatori de resurse;
- parametri importanți;
- decizii interne ale codului;
- dimensiunea listelor sau rezultatelor.

De exemplu, înainte de a căuta un produs în repository, se poate loga:

```
log.debug("Se caută produsul cu id={}", id);
```

1.6.3. INFO

`INFO` este nivelul cel mai potrivit pentru evenimentele normale și importante din aplicație. El descrie faptul că o operație semnificativă a avut loc și, de regulă, s-a încheiat cu succes.

De exemplu:

```
log.info("A fost creat produsul cu id={}", savedProduct.getId());
```

Acest nivel este foarte util deoarece oferă o imagine de ansamblu asupra activității aplicației, fără a produce prea mult zgomot.

1.6.4. WARN

`WARN` se folosește atunci când apare o situație neobișnuită sau nedorită, dar pe care aplicația o poate gestiona controlat. Un exemplu foarte bun în aplicația actuală este cazul în care un produs nu există. Din punctul de vedere al clientului, rezultatul este `404 Not Found`, însă din punctul de vedere al logging-ului nu este neapărat o eroare internă gravă; este mai degrabă un avertisment care merită consemnat.

Exemplu:

```
log.warn("Produsul cu id={} nu a fost găsit.", id);
```

1.6.5. ERROR

`ERROR` este rezervat pentru erori reale: excepții neașteptate, probleme care împiedică finalizarea normală a unei operații, situații care indică un defect sau o problemă semnificativă. De regulă, la acest nivel este util să fie logată și excepția, astfel încât să existe stacktrace în jurnal.

Exemplu:

```
log.error("A apărut o eroare internă neașteptată.", ex);
```

1.7. Cum alegem nivelul potrivit

Alegerea nivelului corect este importantă, deoarece un logging bun nu înseamnă doar „multe mesaje”, ci mesaje relevante, la severitatea potrivită.

În mod practic, se pot urma următoarele reguli:

- se folosește `INFO` pentru operațiile normale și importante;

- se folosește `DEBUG` pentru detalii utile programatorului;
- se folosește `WARN` pentru situații controlate, dar anormale;
- se folosește `ERROR` pentru erori reale;
- `TRACE` se păstrează doar pentru situații foarte detaliate.

O eroare frecventă este folosirea excesivă a lui `ERROR`. De exemplu, dacă produsul nu există, nu înseamnă neapărat că aplicația are o problemă internă. Mai corect este să tratăm această situație ca pe un `WARN`, iar `ERROR` să fie rezervat pentru excepțiile neașteptate și defectele reale.

1.8. Bune practici de logging

1.8.1. Logăm acolo unde informația are sens

Logurile trebuie plasate în punctele în care aduc claritate asupra execuției. Nu este util ca aceeași informație să fie repetată inutil în toate straturile aplicației. De exemplu, controller-ul poate loga intrarea într-un endpoint, service-ul poate loga operația de business, iar handler-ul global poate loga eroarea finală.

1.8.2. Mesajele trebuie să fie clare

Un mesaj bun trebuie să spună concis ce s-a întâmplat și, dacă este cazul, asupra cărei resurse. De exemplu:

```
log.info("Se șterge produsul cu id={}", id);
```

este mult mai util decât un mesaj vag de tipul:

```
log.info("Am ajuns aici");
```

1.8.3. Folosim parametri, nu concatenare

Este recomandat să se folosească stilul parametrizat:

```
log.debug("Se caută produsul cu id={}", id);
```

și nu concatenări precum:

```
log.debug("Se caută produsul cu id=" + id);
```

Forma parametrizată este mai curată, mai eficientă și reprezintă stilul standard în logging-ul modern Java.

1.8.4. Nu logăm date sensibile

În aplicațiile reale trebuie evitată logarea parolelor, token-urilor, datelor personale sau altor informații sensibile.

1.8.5. Nu logăm excesiv

Prea multe loguri pot fi aproape la fel de dăunătoare ca lipsa lor. Dacă fiecare metodă și fiecare detaliu minor generează mesaje `INFO`, consola devine greu de urmărit. Scopul este

să existe suficiente loguri pentru a înțelege execuția, dar nu atât de multe încât informația importantă să se piardă.

1.8.6. Mesajele trebuie să fie consistente

Este bine ca stilul mesajelor să fie unitar în întreaga aplicație. De exemplu, toate mesajele despre produse pot folosi aceeași formulare și aceeași convenție pentru ID-uri.

1.8.7. Excepțiile trebuie logate acolo unde are sens

Dacă o excepție este tratată centralizat în `@RestControllerAdvice`, de multe ori este suficient să fie logată acolo. Dacă aceeași excepție este logată și în service, și în controller, și în handler, apare duplicare inutilă. Alegerea locului corect de logare este, așadar, importantă.

1.9. Crearea loggerelor în mod clasic

În stilul clasic, loggerul este declarat explicit într-o clasă folosind API-ul SLF4J. Forma standard este:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ProductService {

    private static final Logger log =
            LoggerFactory.getLogger(ProductService.class);
}
```

Avantajele stilului clasic sunt:

- este ușor de înțeles mecanismul;
- nu depinde de Lombok;
- este foarte clar ce obiect este folosit pentru logare.

Dezavantajul principal este boilerplate-ul. Dacă multe clase au nevoie de logger, aceeași declarație trebuie repetată de foarte multe ori. [1]

1.10. Crearea loggerelor cu Lombok

În proiectul dezvoltat anterior se utilizează deja Lombok pentru generarea automată a getterelor, setterelor, constructorilor și builderelor, iar printre adnotările utile prezentate apare și `@Slf4j`. Din acest motiv, utilizarea Lombok pentru logging este o continuare naturală și coerentă a stilului deja adoptat în aplicație.

Cu Lombok, loggerul se declară mult mai simplu:

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
@Service
```

```
public class ProductService {
}
```

Adnotarea `@Slf4j` face ca Lombok să genereze automat un logger SLF4J cu numele `log`. Astfel, clasa poate folosi direct:

```
log.info(...);
log.debug(...);
log.warn(...);
log.error(...);
```

Avantajele acestei abordări sunt:

- se elimină codul repetitiv;
- clasele devin mai curate;
- se păstrează coerența cu restul folosirii Lombok din proiect;
- este o practică foarte răspândită în proiectele Spring Boot moderne.

Ca dezavantaj, pentru începători mecanismul poate părea puțin „magic”, deoarece loggerul nu este declarat explicit în codul vizibil, ci este generat la compilare.

1.10.1. Echivalența dintre stilul clasic și stilul cu Lombok

Stilul clasic:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class ProductController {

    private static final Logger log =
            LoggerFactory.getLogger(ProductController.class);
}
```

Stilul cu Lombok:

```
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class ProductController {
}
```

Din punct de vedere funcțional, cele două variante urmăresc același scop. Diferența este că în varianta cu Lombok codul boilerplate nu mai este scris manual, ci este generat automat.

1.11. Diferite tipuri de loggere

În mod curent, în aplicațiile Spring Boot se folosește loggerul bazat pe SLF4J. Totuși, este util să se menționeze că există și alte variante.

1.11.1. Logger clasic SLF4J

Este cel mai frecvent și recomandat în aplicațiile Spring Boot:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
```

1.11.2. Logger generat prin @Slf4j

Acesta este tot un logger SLF4J, dar obținut automat prin Lombok.

1.11.3. Alte adnotări Lombok pentru logging

Lombok oferă și alte adnotări:

```
@Log
@Slf4j
@Log4j2
@CommonsLog
```

Acestea generează loggere pentru diverse API-uri de logging. În contextul Spring Boot standard, `@Slf4j` este în general alegerea cea mai naturală și mai recomandată.

2. Integrarea logging-ului în aplicația REST

În aplicația dezvoltată anterior există deja o structură foarte bună pentru acest lucru. Controller-ele sunt zvelte și responsabile în principal pentru interacțiunea HTTP, service-ul conține logica de business, iar `GlobalExceptionHandler` centralizează tratarea erorilor. Tocmai această separare face ca logging-ul să poată fi introdus într-un mod clar și coerent.

2.1. Principiul de integrare

Înainte de a adăuga efectiv loguri, este important să stabilim o regulă simplă pentru fiecare strat al aplicației.

2.1.1. Controller

În controller este util să logăm:

- apelarea endpoint-ului;
- parametrii principali din cerere, atunci când sunt relevanți.

Nu este recomandat ca aici să se descrie logica internă a aplicației, deoarece controller-ul trebuie să rămână zvelt.

2.1.2. Service

În service este util să logăm:

- începutul unei operații de business;
- decizii importante;
- finalizarea cu succes a unei operații;
- situații controlate, dar neobișnuite.

Acesta este locul cel mai potrivit pentru cele mai multe loguri utile aplicației.

2.1.3. GlobalExceptionHandler

Aici este util să logăm:

- erorile de validare;
- resursele inexistente;
- excepțiile neașteptate.

Acest punct este important deoarece handler-ul global este locul în care aplicația transformă excepțiile în răspunsuri HTTP uniforme.

2.2. Adăugarea loggerului cu Lombok

Pentru că proiectul folosește deja Lombok, cea mai potrivită abordare este utilizarea adnotării `@Slf4j`. Ea generează automat un logger SLF4J cu numele log.

Pentru a o folosi, este suficient să se adauge importul:

```
import lombok.extern.slf4j.Slf4j;
```

și adnotarea pe clasă:

```
@Slf4j
```

După aceasta, în interiorul clasei se poate folosi direct:

```
log.trace(...);
log.debug(...);
log.info(...);
log.warn(...);
log.error(...);
```

2.3. Logging în ProductController

Controller-ul este primul punct de intrare în aplicație. Din acest motiv, aici este util să se consemneze faptul că un endpoint a fost apelat. Nu este nevoie de foarte multe loguri, ci doar de mesaje clare și scurte.

Varianta completă:

```
package ro.unitbv.restlab.controller;

import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.service.ProductService;
```

```
import java.util.List;

@RestController
@RequestMapping("/api/products")
@RequiredArgsConstructor
@Slf4j
public class ProductController {

    private final ProductService productService;

    @GetMapping
    public List<ProductResponse> getAllProducts() {
        log.info("A fost apelat endpoint-ul GET /api/products");
        return productService.findAll();
    }

    @GetMapping("/{id}")
    public ProductResponse getProductById(@PathVariable Integer id) {
        log.info("A fost apelat endpoint-ul GET /api/products/{}", id);
        return productService.findById(id);
    }

    @PostMapping
    public ResponseEntity<ProductResponse> createProduct(
            @Valid @RequestBody CreateProductRequest request) {

        log.info("A fost apelat endpoint-ul POST /api/products pentru
produsul cu numele={}", request.getName());

        ProductResponse createdProduct = productService.create(request);
        return ResponseEntity.status(HttpStatus.CREATED)
                             .body(createdProduct);
    }

    @PutMapping("/{id}")
    public ProductResponse updateProduct(
            @PathVariable Integer id,
            @Valid @RequestBody UpdateProductRequest request) {

        log.info("A fost apelat endpoint-ul PUT /api/products/{}", id);
        return productService.update(id, request);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
        log.info("A fost apelat endpoint-ul DELETE /api/products/{}", id);
        productService.deleteById(id);
        return ResponseEntity.noContent().build();
    }
}
```

2.3.1. Observații asupra logging-ului din controller

Mesajele din controller sunt la nivel `INFO` deoarece descriu evenimente normale și importante: faptul că un endpoint a fost accesat. Aceste informații sunt utile pentru a urmări ce cereri ajung în aplicație.

Se observă și faptul că sunt logați doar parametrii principali:

- ID-ul produsului;
- numele produsului la creare.

Nu este nevoie să se logheze tot request-ul sau toate câmpurile din el. În aplicațiile reale, această practică trebuie făcută cu grijă, deoarece unele date pot fi sensibile.

2.4. Logging în ProductService

Service-ul este locul cel mai potrivit pentru logurile legate de logica de business. Aici putem urmări:

- începutul unei operații;
- detaliile utile dezvoltatorului;
- finalizarea cu succes;
- situațiile controlate, dar anormale.

Varianta completă:

```
package ro.unitbv.restlab.service;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.mapper.ProductMapper;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;
import java.util.List;

@Service
@RequiredArgsConstructor
@Slf4j
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    public List<ProductResponse> findAll() {
        log.debug("Se solicită lista tuturor produselor.");

        List<ProductResponse> result = productRepository.findAll()
                .stream()
```

```
                .map(productMapper::toResponse)
                .toList();

        log.info("Au fost returnate {} produse.", result.size());
        return result;
    }

    public ProductResponse findById(Integer id) {
        log.debug("Se caută produsul cu id={}", id);

        Product product = productRepository.findById(id)
                .orElseThrow(() -> {
                    log.warn("Produsul cu id={} nu a fost găsit.", id);
                    return new ProductNotFoundException(id);
                });

        log.info("Produsul cu id={} a fost găsit.", id);
        return productMapper.toResponse(product);
    }

    public ProductResponse create(CreateProductRequest request) {
        log.debug("Se creează un produs nou: name={}, price={}, stock={}",
                request.getName(), request.getPrice(), request.getStock());

        Product product = productMapper.toEntity(request);
        Product savedProduct = productRepository.save(product);

        log.info("A fost creat produsul cu id={}", savedProduct.getId());
        return productMapper.toResponse(savedProduct);
    }

    public ProductResponse update(Integer id,
                                  UpdateProductRequest request) {
        log.debug("Se actualizează produsul cu id={}", id);

        Product existingProduct = productRepository.findById(id)
                .orElseThrow(() -> {
                    log.warn(
                   "Produsul cu id={} nu a fost găsit pentru actualizare.",
                   id);
                    return new ProductNotFoundException(id);
                });

        productMapper.updateEntity(existingProduct, request);
        Product updatedProduct = productRepository.save(existingProduct);

        log.info("Produsul cu id={} a fost actualizat.",
                 updatedProduct.getId());

        return productMapper.toResponse(updatedProduct);
    }

    public void deleteById(Integer id) {
        log.debug("Se încearcă ștergerea produsului cu id={}", id);
```

```
            if (!productRepository.existsById(id)) {
                log.warn(
                   "Nu se poate șterge produsul cu id={} deoarece nu există.",
                   id);
                throw new ProductNotFoundException(id);
            }

            productRepository.deleteById(id);
            log.info("Produsul cu id={} a fost șters.", id);
        }
    }
```

2.4.1. Observații asupra logging-ului din service

În această clasă apar trei niveluri importante:

DEBUG

Este folosit pentru începutul operațiilor și pentru detalii utile dezvoltatorului. Exemple:

- „Se caută produsul cu id=...”
- „Se creează un produs nou...”
- „Se încearcă ștergerea produsului...”

Aceste loguri sunt foarte utile în dezvoltare și testare, dar nu sunt întotdeauna necesare în producție.

INFO

Este folosit pentru evenimente normale, finalizate cu succes:

- produs găsit;
- produs creat;
- produs actualizat;
- produs șters;
- numărul de produse returnate.

Aceste mesaje descriu foarte bine activitatea normală a aplicației.

WARN

Este folosit pentru situații anormale, dar controlate:

- produsul nu există;
- produsul nu poate fi șters pentru că nu există.

Acestea nu sunt neapărat defecte interne ale aplicației, dar merită consemnate.

2.4.2. Dublarea logurilor

În varianta de mai sus, o situație precum „produsul nu a fost găsit” este logată în service cu `WARN`, iar apoi aceeași excepție poate fi logată din nou în `GlobalExceptionHandler`. Acest

lucru poate fi util didactic, pentru că arată traseul complet al problemei, dar în proiectele reale trebuie ales cu grijă dacă se dorește această redundanță.

Există două stiluri legitime:

- Stilul 1 – logare și în service, și în handler
  - Avantaj: se vede mai clar unde a apărut situația și unde a fost transformată în răspuns HTTP.
  - Dezavantaj: pot apărea loguri repetitive.
- Stilul 2 – logare doar în handler pentru excepții
  - Avantaj: loguri mai curate și mai puține.
  - Dezavantaj: se pierde uneori contextul exact din punctul în care a fost decisă aruncarea excepției.

Pentru un laborator didactic, prima variantă este acceptabilă, tocmai pentru a face fluxul mai explicit.

2.5. Logging în GlobalExceptionHandler

Această clasă este deja prezentă în aplicația anterioară și are rolul de a transforma excepțiile în răspunsuri HTTP uniforme. Din acest motiv, este un loc excelent pentru logarea erorilor și a situațiilor anormale.

Varianta completă:

```
package ro.unitbv.restlab.exception;

import jakarta.servlet.http.HttpServletRequest;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.MethodArgumentNotValidException;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.RestControllerAdvice;
import ro.unitbv.restlab.dto.response.ErrorResponse;
import ro.unitbv.restlab.dto.response.ValidationExceptionResponse;

import java.time.LocalDateTime;
import java.util.LinkedHashMap;
import java.util.Map;

@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(
            ProductNotFoundException ex,
            HttpServletRequest request) {

        log.warn("Resursă inexistentă pentru path={}: {}",
                request.getRequestURI(), ex.getMessage());
```

```
    ErrorResponse response = ErrorResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.NOT_FOUND.value())
            .error(HttpStatus.NOT_FOUND.getReasonPhrase())
            .message(ex.getMessage())
            .path(request.getRequestURI())
            .build();

    return ResponseEntity.status(HttpStatus.NOT_FOUND).body(response);
}

@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ValidationExceptionResponse>
                            handleValidationException(
                                    MethodArgumentNotValidException ex,
                                    HttpServletRequest request) {

    Map<String, String> validationErrors = new LinkedHashMap<>();
    ex.getBindingResult().getFieldErrors()
            .forEach(error ->
                    validationErrors.put(error.getField(),
                                         error.getDefaultMessage()));

    log.warn("Eroare de validare pentru path={}: {}",
             request.getRequestURI(), validationErrors);

    ValidationExceptionResponse response =
            ValidationExceptionResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.BAD_REQUEST.value())
            .error(HttpStatus.BAD_REQUEST.getReasonPhrase())
            .validationErrors(validationErrors)
            .path(request.getRequestURI())
            .build();
    return ResponseEntity.badRequest().body(response);
}

@ExceptionHandler(Exception.class)
public ResponseEntity<ErrorResponse> handleGenericException(
        Exception ex,
        HttpServletRequest request) {

    log.error("Eroare internă neașteptată pentru path={}",
              request.getRequestURI(), ex);

    ErrorResponse response = ErrorResponse.builder()
            .timestamp(LocalDateTime.now())
            .status(HttpStatus.INTERNAL_SERVER_ERROR.value())
            .error(HttpStatus.INTERNAL_SERVER_ERROR.getReasonPhrase())
            .message("A aparut o eroare interna.")
            .path(request.getRequestURI())
            .build();

    return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR)
```

```
                .body(response);
        }
    }
```

2.5.1. Observații asupra logging-ului din handler

Aici apare o diferențiere foarte clară:

WARN pentru erori controlate

- `ProductNotFoundException`
- `MethodArgumentNotValidException`

Aceste situații nu indică o defectare internă gravă a aplicației, ci mai degrabă:

- resursa cerută nu există;
- datele primite nu respectă regulile de validare.

ERROR pentru erori neașteptate

În handler-ul generic, orice excepție necapturată explicit este tratată ca o problemă serioasă și logată la nivel `ERROR`, împreună cu obiectul excepției. Acest lucru este important deoarece astfel se obține și stacktrace-ul.

2.6. Varianta clasică a unui logger, pentru comparație

Mai jos este un exemplu de `ProductService` fără Lombok pentru logger.

```
package ro.unitbv.restlab.service;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Service;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.mapper.ProductMapper;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;

import java.util.List;

@Service
public class ProductService {

    private static final Logger log =
            LoggerFactory.getLogger(ProductService.class);

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    public ProductService(ProductRepository productRepository,
                          ProductMapper productMapper) {
```

```
        this.productRepository = productRepository;
        this.productMapper = productMapper;
    }

    public List<ProductResponse> findAll() {
        log.debug("Se solicită lista tuturor produselor.");

        List<ProductResponse> result = productRepository.findAll()
                .stream()
                .map(productMapper::toResponse)
                .toList();

        log.info("Au fost returnate {} produse.", result.size());
        return result;
    }

    public ProductResponse findById(Integer id) {
        log.debug("Se caută produsul cu id={}", id);

        Product product = productRepository.findById(id)
                .orElseThrow(() -> {
                    log.warn("Produsul cu id={} nu a fost găsit.", id);
                    return new ProductNotFoundException(id);
                });

        log.info("Produsul cu id={} a fost găsit.", id);
        return productMapper.toResponse(product);
    }
}
```

Această variantă este utilă pentru a arăta clar mecanismul de creare a loggerului. Din punct de vedere funcțional, diferența față de `@Slf4j` este minimă; cea mai mare diferență este cantitatea de boilerplate.

2.7. Configurarea nivelurilor în application.properties

Pentru a observa efectiv logurile, trebuie configurat nivelul de logare. [1]

Variantă recomandată pentru laborator este:

```
spring.application.name=restlab-l6

spring.datasource.url=jdbc:h2:mem:restlab;MODE=MySQL;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.open-in-view=false
```

```
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

logging.level.root=INFO
logging.level.ro.unitbv.restlab=DEBUG
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
```

2.7.1. Semnificația configurației

`logging.level.root=INFO` stabilește nivelul implicit pentru întreaga aplicație.

`logging.level.ro.unitbv.restlab=DEBUG` permite afișarea mesajelor `DEBUG` pentru pachetul aplicației noastre.

`logging.pattern.console=...` controlează formatul afișat în consolă.

Această combinație este foarte potrivită pentru laborator, deoarece:

- nu inundă consola cu prea multe mesaje din framework;
- arată totuși suficient de clar logurile proprii.

2.8. Exemple de scenarii și logurile corespunzătoare

2.8.1. Scenariul 1 – listarea tuturor produselor

Cerere:

```
GET /api/products
```

Posibile loguri:

```
2026-03-29 11:10:15 INFO  ProductController - A fost apelat endpoint-ul GET /api/products
2026-03-29 11:10:15 DEBUG ProductService    - Se solicită lista tuturor produselor.
2026-03-29 11:10:15 INFO  ProductService    - Au fost returnate 5 produse.
```

Acest scenariu ilustrează bine:

- INFO în controller pentru apelul endpoint-ului;
- DEBUG în service pentru începutul operației;
- INFO în service pentru rezultatul final.

2.8.2. Scenariul 2 – căutarea unui produs existent

Cerere:

```
GET /api/products/1
```

Loguri:

```
2026-03-29 11:11:03 INFO  ProductController - A fost apelat endpoint-ul GET /api/products/1
2026-03-29 11:11:03 DEBUG ProductService    - Se caută produsul cu id=1
2026-03-29 11:11:03 INFO  ProductService    - Produsul cu id=1 a fost găsit.
```

2.8.3. Scenariul 3 – căutarea unui produs inexistent

Cerere:

```
GET /api/products/999
```

Loguri:

```
2026-03-29 11:12:41 INFO  ProductController      - A fost apelat endpoint-ul GET
/api/products/999
2026-03-29 11:12:41 DEBUG ProductService         - Se caută produsul cu id=999
2026-03-29 11:12:41 WARN  ProductService         - Produsul cu id=999 nu a fost găsit.
2026-03-29 11:12:41 WARN  GlobalExceptionHandler - Resursă inexistentă pentru
path=/api/products/999: Produsul cu id-ul 999 nu a fost gasit.
```

Acest exemplu este foarte bun pentru discuția despre traseul unei excepții:

- cererea intră în controller;
- service-ul încearcă să găsească resursa;
- service-ul constată lipsa ei și aruncă excepția;
- handler-ul global transformă excepția în răspuns 404.

2.8.4. Scenariul 4 – creare reușită

Cerere:

```
POST /api/products
```

Body:

```
{
  "name": "Laptop",
  "price": 3500.0,
  "stock": 10
}
```

Loguri:

```
2026-03-29 11:14:08 INFO  ProductController - A fost apelat endpoint-ul POST /api/products
pentru produsul cu numele=Laptop
2026-03-29 11:14:08 DEBUG ProductService    - Se creează un produs nou: name=Laptop,
price=3500.0, stock=10
2026-03-29 11:14:08 INFO  ProductService    - A fost creat produsul cu id=7
```

2.8.5. Scenariul 5 – creare cu date invalide

Cerere:

```
POST /api/products
```

Body:

```
{
  "name": "",
  "price": 0,
  "stock": -2
}
```

Loguri:

```
2026-03-29 11:15:27 INFO  ProductController      - A fost apelat endpoint-ul POST
/api/products pentru produsul cu numele=
2026-03-29 11:15:27 WARN  GlobalExceptionHandler - Eroare de validare pentru
path=/api/products: {name=Numele produsului este obligatoriu., price=Pretul trebuie sa fie mai
mare decat 0., stock=Stocul nu poate fi negativ.}
```

Aici se poate discuta și despre faptul că uneori e preferabil ca în controller să nu se logheze direct anumite câmpuri din request dacă există riscul ca acestea să fie invalide sau sensibile.

2.8.6. Scenariul 6 – eroare internă neașteptată

Dacă într-o metodă din service ar apărea o excepție necontrolată, de tip `NullPointerException`, `IllegalStateException` sau alta similară, handler-ul generic ar produce un log de forma:

```
2026-03-29 11:17:01 ERROR GlobalExceptionHandler - Eroare internă neașteptată pentru
path=/api/products/3
java.lang.NullPointerException: ...
    at ...
```

Acest tip de log este foarte important deoarece furnizează contextul și stacktrace-ul necesar depanării.

3. Colectarea și analiza logurilor cu Elastic Stack (ELK) folosind Docker

Pentru analiza logurilor într-o aplicație modernă nu este suficientă vizualizarea acestora în consolă sau în fișiere locale. Este necesar un sistem care să permită:

- colectarea automată a logurilor;
- stocarea eficientă;
- căutarea rapidă;
- filtrarea și analiza.

Unul dintre cele mai răspândite stack-uri utilizate în industrie pentru acest scop este **Elastic Stack (ELK)**, compus din:

- Elasticsearch – motor de stocare și căutare;
- Kibana – interfață web pentru explorare;
- Filebeat – colector de loguri.

În această configurație, aplicația Spring Boot scrie logurile într-un fișier, Filebeat le citește și le trimite către Elasticsearch, iar Kibana permite căutarea și analiza acestora.

Se va configura un mediu minimal pentru colectarea logurilor și interogarea acestora folosind o interfață web.

3.1. Fluxul de loguri

Fluxul complet este:

```
aplicație → fișier log → Filebeat → Elasticsearch → Kibana
```

- aplicația generează loguri;
- Filebeat citește fișierul de log;
- logurile sunt trimise către Elasticsearch;
- Kibana permite interogarea și vizualizarea acestora.

3.2. Configurarea aplicației Spring Boot

Pentru ca logurile să poată fi colectate, aplicația trebuie să le scrie într-un fișier, iar pentru ca logurile să fie interpretate corect, cu toate elementele lor (timestamp, nivel, mesaj), este indicat ca să fie salvate în format JSON. De aceea trebuie făcute mai multe modificări de configurație. [1]

În `pom.xml` se va adăuga dependența:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>8.0</version>
</dependency>
```

Configurația componentei `logback` se va face separat, astfel că setările referitoare la formatul logurilor din `application.properties` nu se vor mai aplica.

Se va crea fișierul `logback-spring.xml` în directorul de resurse al aplicației, cu conținutul:

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>
               %d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>

    <appender name="FILE_JSON" class="ch.qos.logback.core.FileAppender">
        <file>logs/application.log</file>
        <append>true</append>

        <encoder class="net.logstash.logback.encoder.LogstashEncoder"/>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="FILE_JSON"/>
    </root>

    <logger name="ro.unitbv.restlab" level="DEBUG"/>
</configuration>
```

Fișierul de configurare va determina crearea a două *appendere*, unul pentru afișarea în consolă și al doilea – pentru salvarea în fișierul `logs/application.log`.

După pornirea aplicației, trebuie verificat că există fișierul:

```
logs/application.log
```

Acesta va fi citit de Filebeat.

3.3. Structura fișierelor pentru Docker

Se creează un director, de exemplu:

```
logging/
 ├── docker-compose.yml
 └── filebeat.yml
```

3.4. Configurarea Docker

3.4.1. Fișierul docker-compose.yml

```
version: '3.8'

services:

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:8.13.0
    container_name: elk-elasticsearch
    environment:
      - discovery.type=single-node
      - xpack.security.enabled=false
    ports:
      - "9200:9200"
    restart: unless-stopped

  kibana:
    image: docker.elastic.co/kibana/kibana:8.13.0
    container_name: elk-kibana
    ports:
      - "5601:5601"
    environment:
      - ELASTICSEARCH_HOSTS=http://elasticsearch:9200
    depends_on:
      - elasticsearch
    restart: unless-stopped

  filebeat:
    image: docker.elastic.co/beats/filebeat:8.13.0
    container_name: elk-filebeat
    user: root
    volumes:
      - ./filebeat.yml:/usr/share/filebeat/filebeat.yml
      - ../logs:/logs
    depends_on:
      - elasticsearch
    restart: unless-stopped
```

Explicația configurației

- **Elasticsearch** rulează în mod single-node (simplificat pentru laborator);
- securitatea este dezactivată pentru a evita configurări suplimentare;
- **Kibana** se conectează la Elasticsearch prin numele serviciului `elasticsearch`;
- **Filebeat**:
  - primește configurația din `filebeat.yml`;
  - montează directorul local `logs` (unde aplicația scrie logurile).

3.4.2. Configurarea Filebeat - fișierul filebeat.yml

```
filebeat.inputs:
  - type: filestream
    enabled: true
    paths:
      - /logs/application.log
    parsers:
      - ndjson:
          target: ""
          overwrite_keys: true

output.elasticsearch:
  hosts: ["http://elasticsearch:9200"]

setup.kibana:
  host: "http://kibana:5601"
```

3.5. Pornirea sistemului

Se pornește prima dată aplicația, pentru ca ea să creeze directorul `logs` și fișierul de log. Apoi se deschide un terminal în directorul `logging` și se execută:

```
docker compose up -d
```

3.6. Verificarea funcționării

3.6.1. Elasticsearch

```
http://localhost:9200
```

Trebuie să returneze informații JSON despre instanță. De exemplu:

```
{
  "name" : "beceecaec7e8",
  "cluster_name" : "docker-cluster",
  "cluster_uuid" : "ZT_QuPK6RfiJB60Mm4scXg",
  "version" : {
    "number" : "8.13.0",
    "build_flavor" : "default",
    "build_type" : "docker",
    "build_hash" : "09df99393193b2c53d92899662a8b8b3c55b45cd",
    "build_date" : "2024-03-22T03:35:46.757803203Z",
    "build_snapshot" : false,
```

```
    "lucene_version" : "9.10.0",
    "minimum_wire_compatibility_version" : "7.17.0",
    "minimum_index_compatibility_version" : "7.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

3.6.2. Kibana

```
http://localhost:5601
```

Se așteaptă câteva secunde până la inițializare. Ar trebui să apară o imagine asemănătoare cu cea din Figura 3. Se alege *Explore my own*.

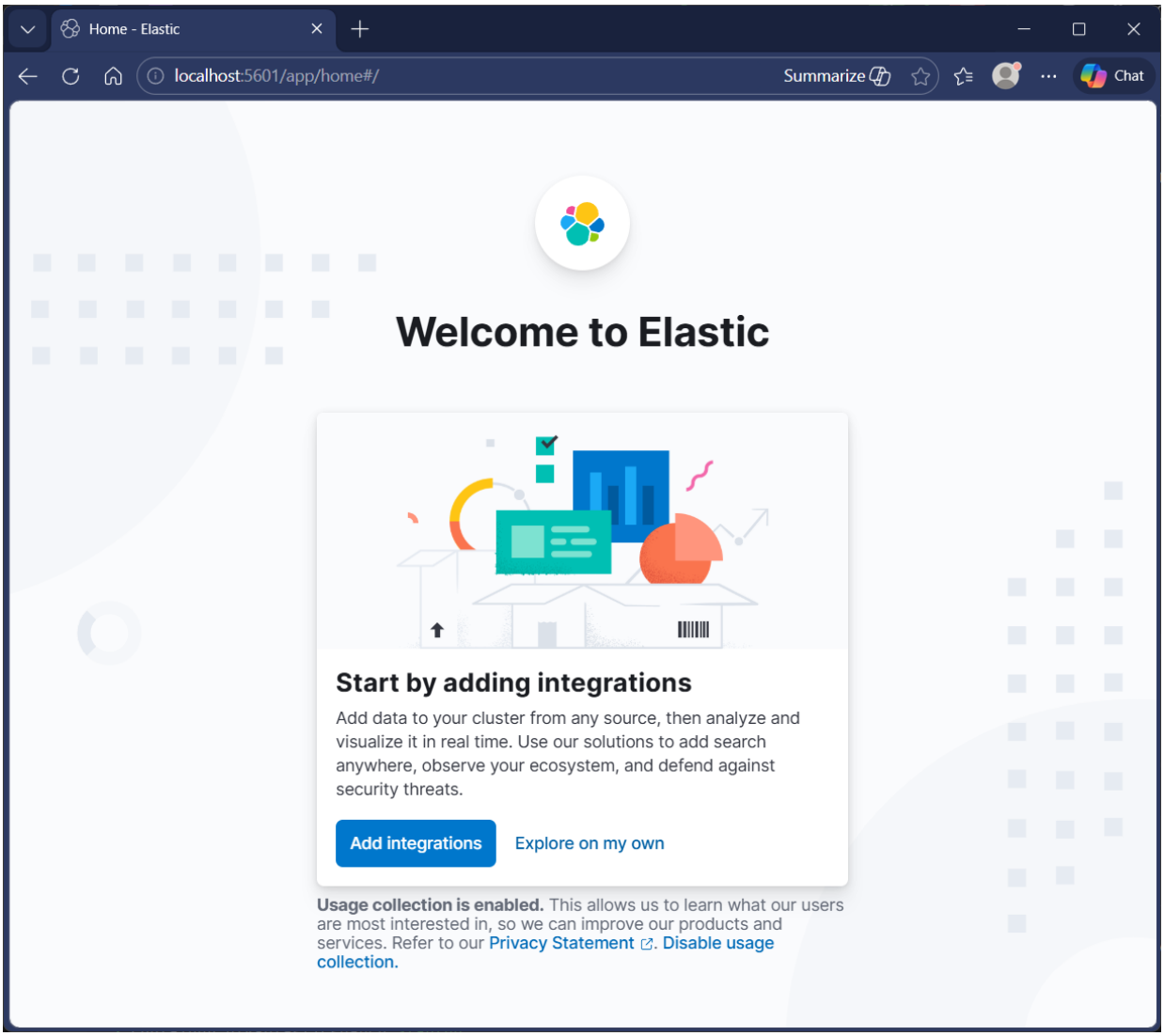


Figura 3 - Pagina de întâmpinare la prima pornire a Kibana (Elastic)

3.7. Configurarea inițială în Kibana/Elastic

La prima accesare:

- se accesează meniul **`Analytics`** → **`Discover`** (sub *hamburger menu* – butonul cu 3 linii de sub sigla *elastic*);

- se creează un index pattern (pe butonul *Create data view*):

  `filebeat-*`

- se selectează câmpul marcă de timp (de regulă `@timestamp`).

După salvare va apare pagina în care putem vizualiza logurile.

3.8. Generarea de loguri

Se execută apeluri către aplicația REST:

```
GET /api/products
POST /api/products
GET /api/products/999
```

Acestea vor genera loguri în fișierul:

```
logs/application.log
```

Filebeat le va trimite automat către Elasticsearch.

3.9. Căutarea logurilor în Kibana

În **`Analytics`** → **`Discover`**, din lista de `Available fields` se vor adăuga câmpurile `level` (nivelul de log) și `message` (este de fapt textul trimis de aplicați în log) (vezi Figura 4 și Figura 5).

Tabelul se va actualiza și va include și aceste câmpuri (Figura 6).

Acum se pot realiza căutări precum:

```
Produsul
```

sau:

```
ERROR
```

sau:

```
WARN
```

Se pot aplica filtre:

- după severitate/nivel (`INFO`, `WARN`, `ERROR`) (Figura 7);
- după interval de timp;
- după text.

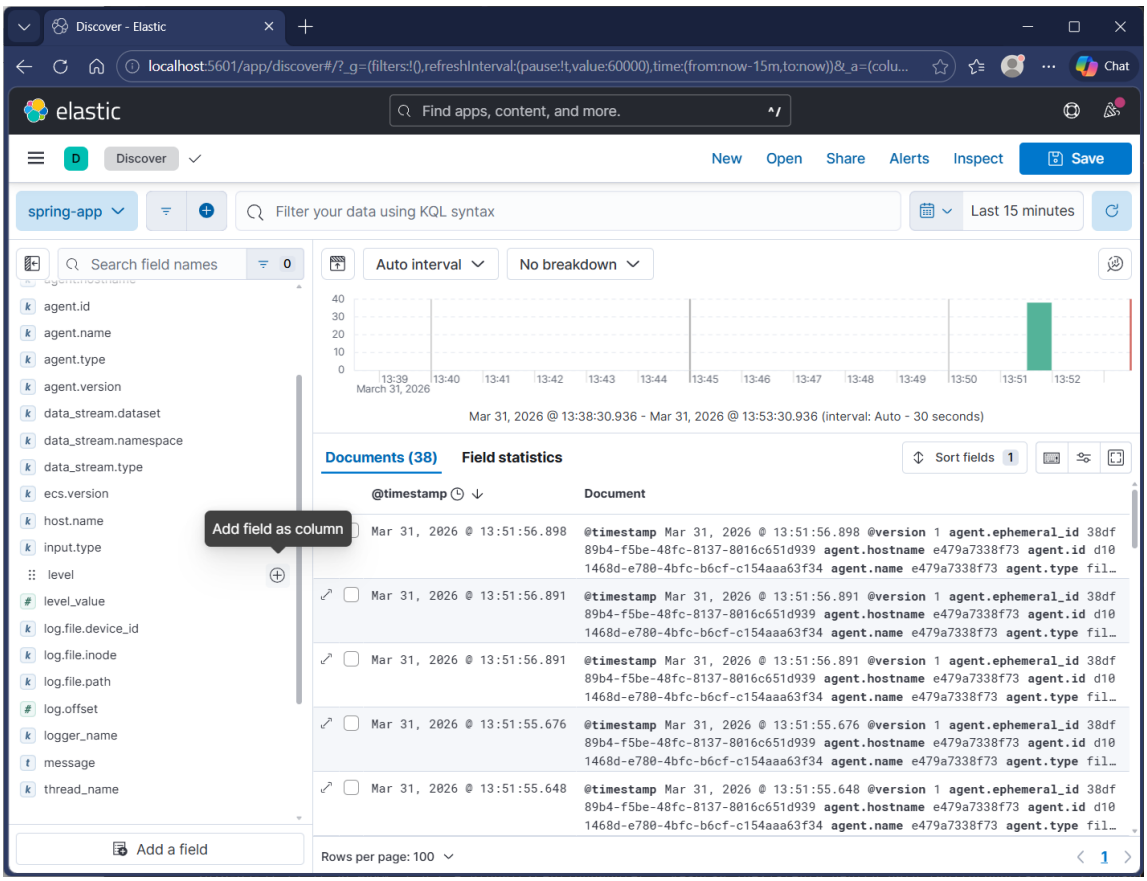


Figura 4 - Adăugarea câmpului "level" în tabelul afișat de pagina Discover a Kibana

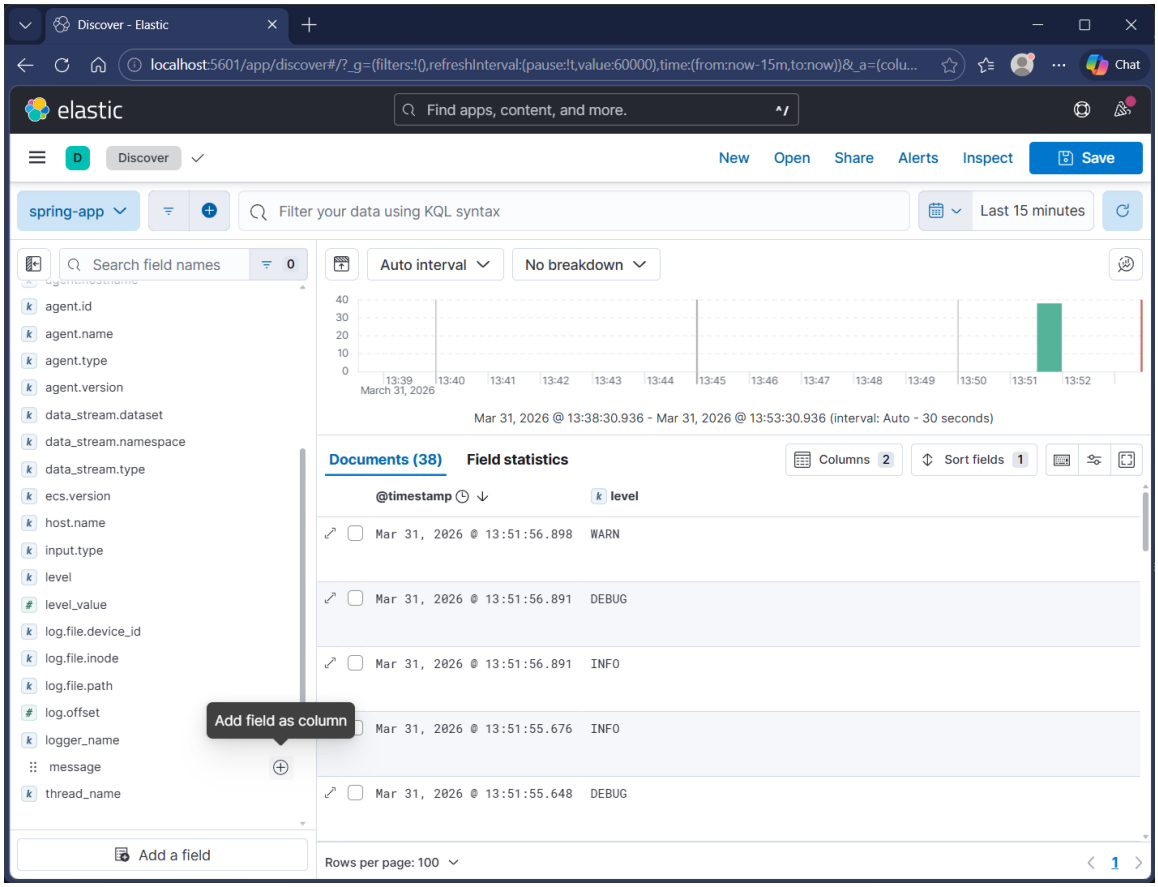


Figura 5 - Adăugarea câmpului "message" în tabelul afișat de pagina Discover a Kibana

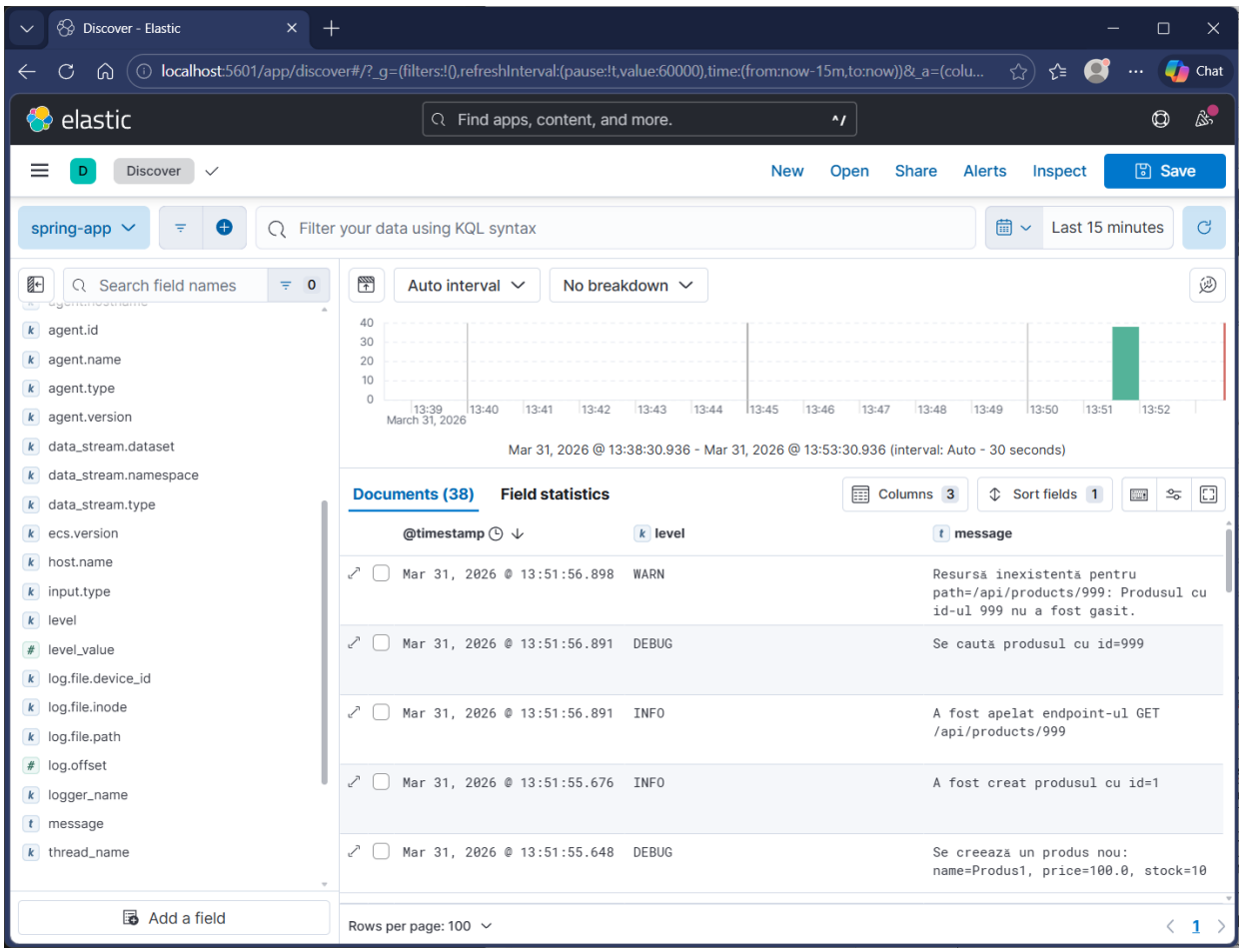


Figura 6 - Pagina Discover afișează mesajele de log

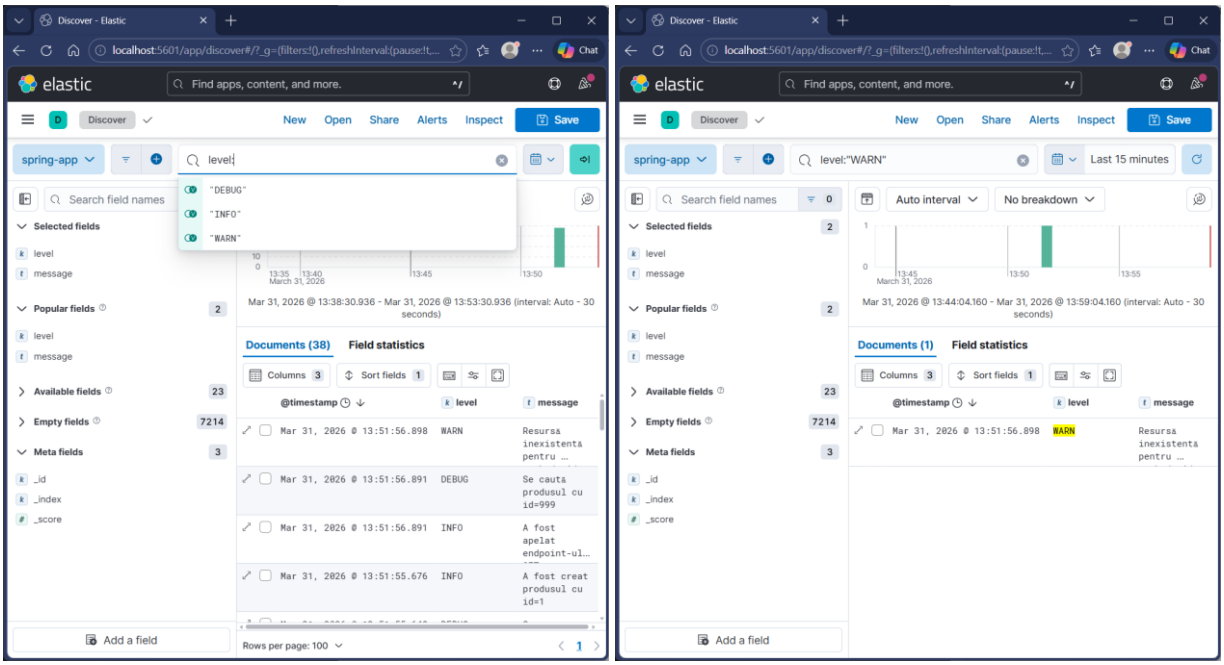


Figura 7 - Filtrarea logurilor după severitate/nivel

3.10. Probleme frecvente

- fișierul de log nu există → Filebeat nu citește nimic;
- calea din `filebeat.yml` este greșită;

- directorul `logs` nu este montat corect;
- Kibana nu are index pattern configurat;
- Elasticsearch nu este încă pornit complet (necesită câteva secunde).

4. Metrici în aplicații Spring Boot

După introducerea logging-ului, aplicația poate descrie evenimente individuale: apeluri de endpoint-uri, operații de business, erori. Totuși, pentru a evalua comportamentul global al sistemului este necesară o abordare diferită, bazată pe măsurători agregate. [1, 23]

Metricile oferă această perspectivă. Ele permit observarea sistemului din punct de vedere cantitativ: frecvență, durată, volum. În absența metricilor, răspunsul la întrebări precum „cât de des apare o eroare" sau „cât durează o operație" ar necesita analiză manuală a logurilor, ceea ce nu este scalabil.

În consecință, metricile sunt esențiale pentru:

- monitorizare;
- analiză de performanță;
- identificarea tendințelor;
- detecția problemelor recurente.

4.1. Definiția unei metrici

O metrică reprezintă o valoare numerică asociată unui moment în timp și, opțional, unui set de etichete (`tags`). Prin colectarea acestor valori în timp se obține o serie de timp (*time series*).

Exemplu conceptual:

```
products.created.count = 120
```

Această valoare indică numărul total de produse create de la pornirea aplicației până în momentul curent.

4.2. Diferența dintre metrici și loguri

Logurile și metricile nu sunt alternative, ci complementare.

Un log descrie un eveniment individual:

```
Produsul cu id=5 a fost creat
```

O metrică descrie comportamentul agregat:

```
120 produse create în ultimele 10 minute
```

Logurile sunt utilizate pentru diagnostic detaliat. Metricile sunt utilizate pentru analiză și monitorizare.

4.3. Infrastructura metricilor în Spring Boot

În Spring Boot, metricile sunt implementate prin două componente:

4.3.1. Micrometer

Micrometer este biblioteca care oferă API-ul pentru definirea și înregistrarea metricilor. Ea este independentă de sistemul de monitorizare utilizat. [1]

4.3.2. Spring Boot Actuator

Actuator expune metricile prin endpoint-uri HTTP. Aceste endpoint-uri pot fi accesate direct sau pot fi consumate de sisteme externe (de exemplu Prometheus).

4.4. Configurarea proiectului

4.4.1. Adăugarea dependenței

În fișierul `pom.xml` trebuie adăugată dependența:

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

Aceasta activează infrastructura de bază pentru metrici.

4.4.2. Configurarea Actuator

În fișierul `application.properties` trebuie adăugate următoarele:

```
management.endpoints.web.exposure.include=health,info,metrics
management.endpoint.health.show-details=always
```

4.4.3. Pornirea aplicației și verificarea

După pornirea aplicației, se verifică:

```
GET http://localhost:8080/actuator/metrics
```

Acest endpoint trebuie să returneze lista metricilor disponibile.

De exemplu, pentru aplicația REST obținem:

```
{
  "names": [
    "application.ready.time",
    "application.started.time",
    "disk.free",
    "disk.total",
    "executor.active",
    "executor.completed",
    "executor.pool.core",
    "executor.pool.max",
    "executor.pool.size",
    "executor.queue.remaining",
```

```
      "executor.queued",
      "hikaricp.connections",
      "hikaricp.connections.acquire",
      "hikaricp.connections.active",
      "hikaricp.connections.creation",
      "hikaricp.connections.idle",
      "hikaricp.connections.max",
      "hikaricp.connections.min",
      "hikaricp.connections.pending",
      "hikaricp.connections.timeout",
      "hikaricp.connections.usage",
      "http.server.requests.active",
      "jdbc.connections.active",
      "jdbc.connections.idle",
      "jdbc.connections.max",
      "jdbc.connections.min",
      "jvm.buffer.count",
      "jvm.buffer.memory.used",
      "jvm.buffer.total.capacity",
      "jvm.classes.loaded",
      ...
    ]
}
```

4.5. Tipuri fundamentale de metrici

În Micrometer există trei tipuri esențiale relevante pentru acest laborator: **Counter**, **Timer** și **Gauge**. [1]

4.5.1. Counter

Un counter este o metrică care numără evenimente. Valoarea sa crește monoton și nu scade.

Un counter nu reprezintă o valoare instantanee, ci numărul total de evenimente înregistrate.

Exemplu:

```
products.created.count = 50
```

Aceasta înseamnă că au fost create 50 de produse de la pornirea aplicației.

Integrare în aplicație

În `ProductService`, se injectează `MeterRegistry`:

```
import io.micrometer.core.instrument.MeterRegistry;

@RequiredArgsConstructor
@Service
@Slf4j
public class ProductService {
    private final MeterRegistry meterRegistry;
    ...
}
```

În metoda `create`:

```
public ProductResponse create(CreateProductRequest request) {

    Product product = productMapper.toEntity(request);
    Product savedProduct = productRepository.save(product);

    meterRegistry.counter("products.created.count").increment();

    log.info("A fost creat produsul cu id={}", savedProduct.getId());

    return productMapper.toResponse(savedProduct);
}
```

Verificare

```
GET /actuator/metrics/products.created.count
```

4.5.2. Timer

Un timer măsoară durata execuției unei operații. Intern, el colectează:

- numărul de apeluri;
- timpul total;
- statistici (maxim, medie etc.).

Permite evaluarea performanței operațiilor.

Integrare în aplicație

În metoda `findById`:

```
import io.micrometer.core.instrument.Timer;

public ProductResponse findById(Integer id) {

    log.debug("Se caută produsul cu id={}", id);

    return Timer.builder("products.findById.time")
            .register(meterRegistry)
            .record(() -> {
                Product product = productRepository.findById(id)
                        .orElseThrow(() ->
                                new ProductNotFoundException(id));

                log.info("Produsul cu id={} a fost găsit.", id);
                return productMapper.toResponse(product);
            });
}
```

Verificare

```
GET /actuator/metrics/products.findById.time
```

4.5.3. Gauge

Un gauge reprezintă o valoare instantanee, care poate crește sau scădea. Spre deosebire de counter, gauge-ul reflectă starea curentă a sistemului.

Exemple:

- numărul de produse existente;
- numărul de utilizatori conectați;
- dimensiunea unei colecții.

Particularitate importantă

Gauge-ul nu este incrementat manual. El este asociat unei surse de date (de exemplu o metodă sau o variabilă), iar valoarea este citită la momentul colectării.

Integrare în aplicație

În `ProductService`, se poate defini un gauge pentru numărul de produse:

```
import jakarta.annotation.PostConstruct;

@PostConstruct
public void initMetrics() {
    meterRegistry.gauge("products.total.count",
            productRepository,
            repo -> repo.count());
}
```

Verificare

```
GET /actuator/metrics/products.total.count
```

Exemplu de răspuns

```
{
  "name": "products.total.count",
  "measurements": [
    {
      "statistic": "VALUE",
      "value": 3
    }
  ],
  "availableTags": []
}
```

Observații

- Gauge-ul reflectă valoarea curentă, nu istorică.
- Este evaluat la momentul interogării.
- În aplicații reale, se preferă metode eficiente (de exemplu `count()` în loc de `findAll().size()`).

`@PostConstruct` este o adnotare din Jakarta (anterior Java EE) folosită pentru a marca o metodă care trebuie executată automat după ce obiectul a fost creat și toate dependențele au fost injectate de Spring.

Pe scurt:

- metoda adnotată cu `@PostConstruct` este apelată o singură dată, la inițializarea bean-ului;
- este locul potrivit pentru inițializări care depind de alte bean-uri deja injectate;
- nu trebuie să primească parametri și nu returnează nimic.

Exemplu:

```
@PostConstruct
public void init() {
    System.out.println("Bean-ul a fost inițializat");
}
```

În contextul metricilor, este utilă pentru înregistrarea gauge-urilor sau a altor metrici imediat după pornirea aplicației, când toate componentele (ex. repository) sunt deja disponibile.

4.6. Metrici pentru erori

În `GlobalExceptionHandler`:

```
@RequiredArgsConstructor
@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    private final MeterRegistry meterRegistry;

    @ExceptionHandler(ProductNotFoundException.class)
    public ResponseEntity<ErrorResponse> handleProductNotFound(...) {

        meterRegistry.counter("products.notfound.count").increment();

        log.warn("Resursă inexistentă pentru path={}: {}",
                 request.getRequestURI(), ex.getMessage());

        ...
    }
}
```

Această metrică permite determinarea frecvenței erorilor.

Verificare

```
GET /actuator/metrics/products.notfound.count
```

4.7. Vizualizarea metricilor

4.7.1. Direct prin Actuator

Metricile pot fi accesate direct:

```
GET /actuator/metrics
GET /actuator/metrics/products.created.count
```

Această metodă este utilă pentru verificare, dar nu este potrivită pentru monitorizare reală.

4.7.2. Instrumente dedicate

Pentru monitorizare reală se utilizează instrumente dedicate. Pentru exemplul de aplicație REST se va folosi:

- Prometheus[14] (pentru colectare)
- Grafana[15] (petru vizualizare)

4.7.3. Configurare Spring Boot pentru Prometheus

În fișierul `application.properties` trebuie adăugate următoarele:

```
management.endpoints.web.exposure.include=health,info,metrics,prometheus
```

În `pom.xml` trebuie adăugată dependența:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Endpoint-ul pentru accesare:

```
GET /actuator/prometheus
```

Exemplu de fragment de răspuns:

```
# HELP application_ready_time_seconds Time taken for the application to be ready to
service requests
# TYPE application_ready_time_seconds gauge
application_ready_time_seconds{main_application_class="ro.unitbv.restlab.RestLabApp
lication"} 3.682
# HELP application_started_time_seconds Time taken to start the application
# TYPE application_started_time_seconds gauge
application_started_time_seconds{main_application_class="ro.unitbv.restlab.RestLabA
pplication"} 3.641
# HELP executor_active_threads The approximate number of threads that are actively
executing tasks
# TYPE executor_active_threads gauge
executor_active_threads{name="applicationTaskExecutor"} 0.0
```

[14] Prometheus este un sistem open-source de monitorizare și colectare de metrici care stochează serii de timp și permite interogarea și alertarea asupra stării aplicațiilor și infrastructurii.
[15] Grafana este o platformă de vizualizare și analiză care permite construirea de dashboard-uri interactive pe baza datelor din surse precum Prometheus, pentru monitorizarea sistemelor și aplicațiilor.

```
# HELP executor_completed_tasks_total The approximate total number of tasks that
have completed execution
# TYPE executor_completed_tasks_total counter
executor_completed_tasks_total{name="applicationTaskExecutor"} 0.0
# HELP executor_pool_core_threads The core number of threads for the pool
# TYPE executor_pool_core_threads gauge
executor_pool_core_threads{name="applicationTaskExecutor"} 8.0
# HELP executor_pool_max_threads The maximum allowed number of threads in the pool
# TYPE executor_pool_max_threads gauge
executor_pool_max_threads{name="applicationTaskExecutor"} 2.147483647E9
# HELP executor_pool_size_threads The current number of threads in the pool
# TYPE executor_pool_size_threads gauge
executor_pool_size_threads{name="applicationTaskExecutor"} 0.0
# HELP executor_queue_remaining_tasks The number of additional elements that this
queue can ideally accept without blocking
# TYPE executor_queue_remaining_tasks gauge
executor_queue_remaining_tasks{name="applicationTaskExecutor"} 2.147483647E9
...
```

4.8. Utilizarea Docker pentru Prometheus și Grafana

După ce aplicația Spring Boot a fost configurată să expună metrici prin Actuator și a fost configurat și Prometheus (pentru colectarea metricilor), următorul pas constă în folosirea unui sistem extern care să le afișeze într-o formă ușor de urmărit. O soluție standard, gratuită și foarte răspândită este Grafana, care afișează metricile sub formă de grafice și dashboard-uri.

Pentru a simplifica instalarea și rularea acestor componente, se poate utiliza Docker. În acest mod, nu este necesară instalarea manuală a Prometheus și Grafana în sistemul de operare; ele vor rula în containere separate.

Fluxul este următorul:

- aplicația Spring Boot expune metrici la /actuator/prometheus;
- Prometheus interoghează periodic acest endpoint;
- Prometheus stochează valorile colectate;
- Grafana citește datele din Prometheus și construiește dashboard-uri.

Cu alte cuvinte, aplicația nu trimite activ metricile către Prometheus. Prometheus este cel care vine și le „citește” la intervale regulate. Acest model se numește **pull model**.

4.8.1. Structura minimă a fișierelor

Într-un director separat, de exemplu `monitoring`, se pot crea două fișiere:

- `docker-compose.yml`
- `prometheus.yml`

De exemplu:

```
monitoring/
  ├── docker-compose.yml
  └── prometheus.yml
```

Este recomandat ca aceste fișiere să fie ținute separat de codul aplicației, pentru a păstra organizarea clară.

4.8.2. Fișierul docker-compose.yml

Acest fișier descrie containerele care trebuie pornite.

Conținut recomandat:

```
version: '3.8'

services:

  prometheus:
    image: prom/prometheus
    container_name: restlab-prometheus
    ports:
      - "9090:9090"
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    restart: unless-stopped

  grafana:
    image: grafana/grafana
    container_name: restlab-grafana
    ports:
      - "3000:3000"
    restart: unless-stopped
```

În această configurație sunt definite două servicii.

Serviciul `prometheus`

Acest serviciu folosește imaginea oficială `prom/prometheus`. Containerul va porni Prometheus și va expune portul `9090` pe mașina locală.

Liniile:

```
ports:
  - "9090:9090"
```

înseamnă că portul `9090` din container este mapat pe portul `9090` al calculatorului local.

Liniile:

```
volumes:
  - ./prometheus.yml:/etc/prometheus/prometheus.yml
```

înseamnă că fișierul local `prometheus.yml` este montat în container ca fișier de configurare Prometheus.

Serviciul `grafana`

Acest serviciu folosește imaginea oficială `grafana/grafana`. Containerul va porni Grafana și va expune interfața web pe portul `3000`.

Liniile:

```
ports:
  - "3000:3000"
```

face posibil accesul la Grafana prin browser la adresa:

```
http://localhost:3000
```

4.8.3. Fișierul prometheus.yml

Acest fișier spune lui Prometheus de unde trebuie să colecteze metrici.

Conținut recomandat:

```
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: 'spring-app'
    metrics_path: '/actuator/prometheus'
    static_configs:
      - targets: ['host.docker.internal:8080']
```

În cadrul acestui fișier:

```
scrape_interval: 5s
```

Aceasta înseamnă că Prometheus va citi metricile la fiecare 5 secunde.

```
job_name: 'spring-app'
```

Este doar un nume logic pentru sursa de metrici. Poate fi orice nume relevant.

```
metrics_path: '/actuator/prometheus'
```

Aceasta este calea HTTP din aplicația Spring Boot unde sunt expuse metricile.

```
targets: ['host.docker.internal:8080']
```

Aceasta este partea cea mai importantă.

Containerul Prometheus rulează în Docker, iar aplicația Spring Boot rulează, de regulă, direct pe calculatorul local. Din interiorul containerului, `localhost` nu mai înseamnă calculatorul gazdă, ci chiar containerul respectiv. De aceea, nu trebuie folosit, în acest caz:

```
targets: ['localhost:8080']
```

Pentru Docker Desktop pe Windows și macOS, `host.docker.internal` este alias-ul standard pentru calculatorul gazdă. Astfel, Prometheus poate ajunge la aplicația Spring Boot rulată local.

4.8.4. Exercițiu de utilizare

Pasul 1 – pornirea aplicației Spring Boot

Aplicația trebuie pornită normal, din IDE sau din linia de comandă.

Trebuie verificat că funcționează:

```
GET http://localhost:8080/actuator/prometheus
```

Dacă acest endpoint răspunde, configurația Spring Boot este corectă.

Pasul 2 – crearea directorului de monitorizare

Se creează un director, de exemplu:

```
monitoring
```

În acest director se salvează fișierele:

```
docker-compose.yml
prometheus.yml
```

Pasul 3 – deschiderea terminalul în directorul respectiv

De exemplu:

```
cd monitoring
```

Pasul 4 – pornirea containerele

Se execută comanda:

```
docker compose up -d
```

Dacă versiunea Docker folosește vechiul plugin Compose, poate fi necesar:

```
docker-compose up -d
```

Opțiunea `-d` înseamnă că serviciile vor rula în fundal.

Pasul 5 – verificarea serviciilor

Se execută:

```
docker ps
```

Ar trebui să apară două containere:

```
restlab-prometheus
```

```
restlab-grafana
```

Pasul 6 – verificarea Prometheus

După pornire, se accesează în browser:

```
http://localhost:9090
```

Interfața Prometheus trebuie să fie disponibilă.

Pentru a verifica dacă Prometheus colectează metricile aplicației, se poate merge în:

```
Status -> Target health
```

Acolo ar trebui să apară sursa `spring-app` cu starea `UP`.

Dacă starea este `DOWN`, înseamnă că Prometheus nu poate ajunge la aplicația Spring Boot. În această situație trebuie verificat:

- dacă aplicația rulează;
- dacă portul este corect;
- dacă `host.docker.internal:8080` este accesibil;
- dacă endpoint-ul `/actuator/prometheus` este expus.

Pasul 7 – interogarea metricilor în Prometheus

În pagina principală Prometheus există o zonă de interogare (query).

Se pot introduce expresii precum:

```
products_created_count
```

sau, dacă Spring/Micrometer normalizează altfel numele metricii, trebuie folosit exact numele afișat în output-ul de la `/actuator/prometheus`.

De exemplu, dacă în aplicație există:

```
meterRegistry.counter("products.created.count").increment();
```

în Prometheus numele apare de regulă în forma:

```
products_created_count
```

Punctele sunt transformate în underscore.

Pasul 8 – verificarea Grafana

După pornire, se accesează:

```
http://localhost:3000
```

Datele implicite de autentificare sunt:

```
username: admin
password: admin
```

La prima autentificare, Grafana poate cere schimbarea parolei.

Pasul 9 – conectarea Grafana la Prometheus

În Grafana trebuie adăugat un data source.

Pașii sunt:

- se accesează meniul **Connections** sau **Data Sources**;
- se alege **Add connection** sau **Add data source**;
- se selectează **Prometheus**;
- se apasă butonul **Add new data source**;
- la URL se introduce: `http://prometheus:9090`

Este important de observat că aici nu se folosește `localhost`, deoarece Grafana și Prometheus rulează în aceeași rețea Docker și se pot vedea între ele după numele serviciului.

Se salvează configurația și se apasă **Save & Test**.

Dacă totul este corect, Grafana va confirma că data source-ul funcționează.

Pasul 10 – crearea unui dashboard simplu în Grafana

După conectarea la Prometheus, se poate crea un dashboard nou.

Pașii sunt:

- se accesează meniul **Dashboards**;
- se alege **New** -> **New dashboard**;
- se adaugă un panel nou (**Add visualization**);
- se selectează data source-ul Prometheus;
- se introduce o interogare.

Exemple de interogări:

Counter – total produse create

```
products_created_count
```

Aceasta afișează valoarea totală a counterului.

Rata de creare pe minut

```
rate(products_created_count[1m])
```

Aceasta afișează rata de creștere în fereastra ultimului minut.

Gauge – număr curent de produse

```
products_total_count
```

Aceasta afișează valoarea curentă a gauge-ului.

Timer – dacă este expus ca serie compatibilă Prometheus

Pentru timere se pot construi interogări mai avansate, în funcție de seriile exportate de Micrometer.

4.8.5. Probleme frecvente și explicații

Problema 1 – Prometheus nu vede aplicația

Dacă în Prometheus target-ul este `DOWN`, cauza este aproape întotdeauna una dintre următoarele:

- aplicația nu rulează pe portul așteptat;
- endpoint-ul `/actuator/prometheus` nu este expus;
- `host.docker.internal` nu este disponibil în mediul respectiv;
- portul din configurația Prometheus este greșit.

Problema 2 – se folosește localhost:8080 în prometheus.yml

Aceasta este o greșeală frecventă. Din container, localhost nu înseamnă calculatorul gazdă, ci containerul însuși. De aceea, pentru aplicația rulată pe calculatorul local, trebuie folosit `host.docker.internal`.

Problema 3 – metrica nu apare în Prometheus

Trebuie verificat:

- dacă metrica există la `/actuator/metrics`;
- dacă apare și în `/actuator/prometheus`;
- dacă numele este căutat corect, în formatul Prometheus.

Problema 4 – Grafana nu se conectează la Prometheus

În Grafana trebuie folosit URL-ul:

```
http://prometheus:9090
```

și nu `http://localhost:9090`, deoarece Grafana rulează în Docker și trebuie să acceseze serviciul Prometheus prin numele containerului.

4.9. Bune practici pentru metrici

Metricile trebuie să reflecte operații relevante din punct de vedere al business-ului. Nu este recomandată măsurarea excesivă a detaliilor interne. [1]

Numele metricilor trebuie să fie descriptive și stabile. Schimbarea frecventă a acestora afectează dashboard-urile și integrarea cu sistemele externe.

Trebuie evitată utilizarea unui număr mare de valori distincte pentru etichete (tags) (produce cardinalitate mare), deoarece poate duce la probleme de performanță în sistemele de monitorizare.

5. Întrebări recapitulative

1. De ce este potrivit să logăm apelarea endpoint-urilor în controller?
2. De ce este service-ul locul principal pentru logurile legate de business?
3. Care este diferența dintre `DEBUG`, `INFO`, `WARN` și `ERROR` în exemplele de mai sus?
4. Care este avantajul utilizării lui `@Slf4j` într-un proiect care folosește deja Lombok?
5. De ce poate deveni problematică dublarea logurilor pentru aceeași excepție?
6. Ce efect are configurarea `logging.level.ro.unitbv.restlab=DEBUG`?
7. Care este diferența fundamentală dintre logging și metrici și de ce sunt necesare ambele într-o aplicație REST?
8. Ce rol au Spring Boot Actuator și Micrometer în gestionarea metricilor într-o aplicație Spring Boot?
   Ce diferență există între endpoint-urile:

   ```
   /actuator/metrics
   /actuator/prometheus
   ```

   și în ce situații se utilizează fiecare?
9. Ce este un counter și de ce valoarea sa nu trebuie interpretată ca o valoare instantanee?
10. Ce este un timer și ce tipuri de informații poate furniza despre execuția unei metode?
11. Ce este un gauge și care este diferența esențială față de counter în ceea ce privește modul de actualizare a valorii? De ce gauge-ul este evaluat „la citire” și nu prin incrementare explicită?
12. Cum contribuie metricile la observabilitatea aplicației și ce tip de probleme pot fi identificate doar prin analiza acestora?

6. Exercițiu – Analiza execuției aplicației REST prin loguri și metrici (Grafana)

Acest exercițiu urmărește corelarea directă dintre:

- apelurile efectuate asupra aplicației REST;
- logurile generate în aplicație;
- evoluția metricilor vizualizate în Grafana.

Scopul este înțelegerea modului în care acțiunile utilizatorului influențează atât execuția aplicației, cât și valorile metricilor colectate.

1.1. Cerință generală

Se cere realizarea unui document (PDF) în care să fie prezentate, pentru fiecare acțiune efectuată asupra aplicației:

- apelul HTTP (endpoint + payload);
- logurile generate de aplicație;
- valorile metricilor relevante;
- observații privind comportamentul aplicației.

Documentul trebuie să conțină **capturi de ecran sau extrase text** atât din loguri, cât și din Grafana.

6.1. Condiții inițiale

Se presupune că:

- aplicația Spring Boot rulează pe `localhost:8080`;
- logging-ul este configurat (INFO + DEBUG pentru aplicație);
- metricile sunt implementate:
  - `products.created.count`
  - `products.notfound.count`
  - `products.findById.time`
  - `products.total.count`
- Prometheus și Grafana rulează prin Docker;
- în Grafana există un dashboard cu:
  - `products_created_count`
  - `products_total_count`
  - `rate(products_created_count[1m])`

6.2. Scenariu de execuție

Se vor executa, în ordine, următoarele operații:

Pasul 1 – Verificare inițială

Se apelează:

```
GET /api/products
```

Ce trebuie inclus în document:

- logurile generate;
- valoarea inițială a:
  - `products_total_count`
  - `products_created_count` (Grafana sau Actuator)

Pasul 2 – Creare produse

Se creează 3 produse diferite:

```
POST /api/products
```

Exemplu de payload (body):

```
{
  "name": "Produs1",
  "price": 100,
```

```
  "stock": 10
}
```

(se repetă de 3 ori cu valori diferite)

Ce trebuie inclus:

- logurile pentru fiecare apel;
- evoluția:
  - `products_created_count`
  - `products_total_count`
- captură Grafana (counter + gauge)

Pasul 3 – Acces produse existente

Se apelează:

```
GET /api/products/1
GET /api/products/2
```

Ce trebuie inclus:

- logurile generate (DEBUG + INFO);
- observații asupra metricii:
  - `products.findById.time`

Pasul 4 – Acces produs inexistent

Se apelează:

```
GET /api/products/999
```

Ce trebuie inclus:

- logurile din:
  - `ProductService` (`WARN`)
  - `GlobalExceptionHandler` (`WARN`)
- evoluția:
  - `products.notfound.count`

Pasul 5 – Ștergere produs

Se apelează:

```
DELETE /api/products/1
```

Ce trebuie inclus:

- logurile aferente;
- evoluția:
  - `products.total.count`

6.3. Structura documentului predat

Documentul trebuie să fie organizat clar, pe pași:

Pentru fiecare pas:

1. Apel HTTP
2. Loguri relevante
3. Metrici (valori sau grafice)
4. Observații

6.4. Întrebări de reflecție (de inclus la final)

1. De ce `products.created.count` nu scade după ștergere?
2. De ce `products.total.count` reflectă corect starea curentă?
3. Cum se observă diferența dintre counter și gauge în Grafana?
4. Ce tip de informații oferă timerul și de ce nu este vizibil ca un simplu număr?
5. Cum se corelează un `WARN` din loguri cu o creștere a unei metrici?

Laborator 5: Spring Security – autentificare și autorizare într-un API REST

1. Obiectivul laboratorului

În acest laborator se urmărește introducerea noțiunilor fundamentale de **securitate** într-o aplicație Spring Boot care expune un API REST. Punctul de plecare este aplicația construită în laboratoarele anterioare (gestionarea unui catalog de produse), la care se vor adăuga, progresiv, mecanisme de autentificare și de autorizare.

Accentul cade pe înțelegerea modului în care Spring Security funcționează în spatele scenei – adică pe arhitectura internă a filtrelor, a contextului de securitate și a fluxurilor de autentificare – precum și pe alegerea conștientă a mecanismului potrivit pentru fiecare tip de aplicație.

La finalul laboratorului, cititorul ar trebui să fie capabil:

- să explice diferența dintre autentificare și autorizare;
- să descrie rolul lanțului de filtre Spring Security și al contextului de securitate;
- să configureze autentificare de tip HTTP Basic, Form Login și JWT[16];
- să definească un serviciu propriu de utilizatori cu baza de date și un `PasswordEncoder`;
- să aplice reguli de autorizare la nivel de URL și la nivel de metodă;
- să trateze corect CORS, CSRF, sesiunile și erorile de autentificare/autorizare.

2. De ce securitate într-un API REST

Un API REST expus către exterior, chiar și într-o rețea privată, nu poate rămâne deschis oricui. În absența oricărei protecții, orice client care cunoaște URL-ul poate citi, modifica sau șterge resurse. Acest lucru devine critic imediat ce datele gestionate au valoare – prețuri, stocuri, comenzi, utilizatori. [3, 24]

Securitatea unei aplicații web se construiește pe două întrebări complementare:

1. **Autentificare** (engl. *authentication*) – cine face cererea? Cum se demonstrează identitatea celui care cere o resursă?

[16] Un JSON Web Token este un token compact și securizat, folosit pentru a transmite informații între părți (de obicei pentru autentificare și autorizare) sub formă de JSON semnat digital.

2. **Autorizare** (engl. *authorization*) – ce îi este permis să facă entității autentificate? Are ea dreptul să acceseze sau să modifice resursa respectivă?

Aceste două preocupări sunt distincte. Un utilizator poate fi autentificat corect (se știe cine este), dar să nu fie autorizat să acceseze un anumit endpoint (nu are rolul necesar). Confundarea celor două mecanisme este una dintre cele mai frecvente surse de vulnerabilități.

Observație: Spring Security tratează explicit cele două concepte ca lanțuri diferite: autentificarea produce un obiect `Authentication` stocat în `SecurityContext`, iar autorizarea verifică, pe baza acestuia, dacă cererea curentă are voie să continue.

3. Ce este Spring Security și cum funcționează

Spring Security este un framework de securitate pentru aplicații bazate pe Spring. Este conceput pe principiul **filtrelor servlet**[17] și al delegării: în loc ca fiecare controller să verifice manual identitatea și permisiunile, aceste verificări sunt efectuate **înainte** ca cererea HTTP să ajungă la controller, printr-un lanț de filtre care pot accepta, refuza sau îmbogăți cererea. [3, 24]

3.1. Modelul conceptual

Înainte de a intra în detalii tehnice, este util să fie reținute următoarele concepte fundamentale:

- **Principal** – entitatea autentificată (de regulă un utilizator, dar poate fi și un alt serviciu).
- **Credentials** – dovada identității (parolă, token, certificat).
- **Authentication** – un obiect care conține principal-ul, credentials și lista permisiunilor / drepturi de acces (authorities) atribuite acestuia; devine reprezentarea în memorie a identității autentificate pentru durata unei cereri.
- **Authority** – o permisiune atomică, reprezentată ca șir de caractere (de exemplu `ROLE_ADMIN` sau `READ_PRODUCTS`).
- **Role** – o grupare convențională de permisiuni, exprimată printr-un nume care, prin convenția Spring Security, este prefixat cu `ROLE_`.
- **SecurityContext** – containerul care păstrează obiectul `Authentication` pentru firul de execuție curent.

3.2. Arhitectura pe bază de filtre

Într-o aplicație Java EE/Spring, cererile HTTP trec printr-un lanț de filtre servlet *(javax.servlet.Filter)* înainte de a ajunge la dispatcher-ul care le direcționează către controller. Spring Security se integrează în acest mecanism prin două componente cheie:

[17] Un Java Servlet este o componentă software care rulează pe un server web și procesează cereri (de obicei HTTP) de la clienți, generând răspunsuri dinamice.

3.3. DelegatingFilterProxy

Este un filtru servlet obișnuit, înregistrat în containerul web, care nu conține logică proprie. Rolul său este să delege fiecare cerere unui bean Spring denumit, convențional, `springSecurityFilterChain`. În acest fel, lanțul de filtre de securitate se află sub controlul ApplicationContext-ului Spring și poate beneficia de injecție de dependențe, configurație și tratare uniformă a bean-urilor. [3]

3.3.1. FilterChainProxy

Este bean-ul care primește cererea de la `DelegatingFilterProxy` și o trece printr-un lanț ordonat de obiecte de tip `SecurityFilterChain`. Fiecare `SecurityFilterChain` definește un tipar de URL-uri și o listă de filtre care se vor aplica cererilor ce se potrivesc. Astfel, în aceeași aplicație pot coexista reguli diferite pentru, de exemplu, `/api/**` (stateless, cu JWT) și `/admin/**` (cu sesiune și form login).

3.3.2. Filtrele uzuale și ordinea lor

În interiorul unui `SecurityFilterChain`, filtrele sunt ordonate strict. Ordinea contează, pentru că fiecare filtru poate consuma cererea sau o poate adnota cu informații folosite mai departe. Cele mai importante filtre, în ordinea în care acționează, sunt:

1. `SecurityContextHolderFilter` – încarcă contextul de securitate din sesiune (dacă există) la începutul cererii.
2. `LogoutFilter` – tratează cererile de deautentificare.
3. `UsernamePasswordAuthenticationFilter` – procesează cereri de login bazate pe formular (implicit pe `POST /login`).
4. `BasicAuthenticationFilter` – extrage și validează credențialele din antetul `Authorization: Basic ....`
5. `BearerTokenAuthenticationFilter` – extrage și validează tokenuri bearer (de exemplu JWT) din antetul `Authorization: Bearer ....`
6. `ExceptionTranslationFilter` – prinde excepțiile de tip `AuthenticationException` și `AccessDeniedException` ridicate mai jos în lanț și le convertește în răspunsuri HTTP corespunzătoare.
7. `AuthorizationFilter` – aplică regulile de autorizare la nivel de URL chiar înainte ca cererea să ajungă la controller.

Notă importantă: Ordinea filtrelor nu este arbitrară. De exemplu, `AuthorizationFilter` se află la finalul lanțului tocmai pentru că verifică permisiunile numai după ce toate filtrele de autentificare au avut ocazia să populeze SecurityContext-ul. Dacă autentificarea s-ar face după autorizare, cererea ar fi respinsă înainte să se știe cine este utilizatorul.

3.4. SecurityContextHolder și SecurityContext

`SecurityContextHolder` este un container static care păstrează, pentru firul de execuție curent, un obiect de tip `SecurityContext`. Acesta, la rândul său, conține obiectul `Authentication` aferent utilizatorului identificat. Strategia implicită de stocare este

`ThreadLocal`, ceea ce înseamnă că fiecare cerere HTTP are propriul context, izolat de celelalte cereri care rulează în paralel. [3]

La finalul procesării cererii, contextul este curățat automat, astfel încât firul de execuție (care poate fi reutilizat de containerul de servleți) să nu poarte informații reziduale.

Din cod se poate obține utilizatorul autentificat astfel:

```
Authentication auth =
               SecurityContextHolder.getContext().getAuthentication();
String username = auth.getName();
Collection<? extends GrantedAuthority> authorities = auth.getAuthorities();
```

Într-un controller Spring, aceeași informație se poate obține mult mai elegant prin injectarea parametrului `Authentication` sau a lui `Principal` în metoda care tratează cererea.

3.5. Fluxul autentificării

Din perspectivă arhitecturală, autentificarea unei cereri parcurge, indiferent de mecanismul concret, aceeași schemă generală [3]:

1. Un filtru de autentificare extrage credențialele din cerere (antet, formular, cookie, token).
2. Filtrul construiește un obiect `Authentication` neautentificat, purtând doar credențialele primite.
3. Filtrul îl transmite către un `AuthenticationManager`, care este, de regulă, un `ProviderManager`.
4. `ProviderManager` deleagă rând pe rând, către o listă de `AuthenticationProvider`-i. Primul care recunoaște tipul de `Authentication` încearcă să îl valideze.
5. Dacă validarea reușește, provider-ul întoarce un obiect `Authentication` marcat ca autentificat, cu autoritățile (rolurile) aferente.
6. Filtrul plasează obiectul autentificat în `SecurityContextHolder` și lasă cererea să continue.
7. Dacă validarea eșuează, filtrul produce un răspuns de eroare (tipic `401 Unauthorized`) și oprește procesarea.

3.6. AuthenticationProvider, UserDetailsService, PasswordEncoder

Cel mai frecvent `AuthenticationProvider` folosit este `DaoAuthenticationProvider`. Acesta se bazează pe două componente [3]:

- **UserDetailsService** – o interfață funcțională cu o singură metodă, `loadUserByUsername(String username)`, care returnează un `UserDetails` (obiect cu username, parolă codificată și permisiuni) sau aruncă `UsernameNotFoundException`.
- **PasswordEncoder** – o interfață care codifică și verifică parole. Implementarea recomandată este `BCryptPasswordEncoder`.

Fluxul pe care îl execută `DaoAuthenticationProvider` este extrem de simplu de urmărit mental:

1. Preia username-ul din `Authentication` și apelează `userDetailsService.loadUserByUsername(...)`.
2. Compară parola primită cu cea stocată folosind `passwordEncoder.matches(raw, encoded)`.
3. Dacă parolele se potrivesc, construiește un obiect `Authentication` autentificat, purtând permisiunile din `UserDetails`.
4. Dacă nu se potrivesc, aruncă `BadCredentialsException`.

3.7. Fluxul autorizării

După ce cererea este autentificată, `AuthorizationFilter` consultă, pentru URL-ul curent, un `AuthorizationManager` care decide dacă cererea este permisă. [3]

`AuthorizationManager` returnează un obiect `AuthorizationDecision` cu un singur câmp: `granted`. Dacă este `false`, filtrul aruncă `AccessDeniedException`, care va fi tradusă de `ExceptionTranslationFilter` într-un răspuns `403 Forbidden`.

3.8. Aplicația de plecare

Se pornește de la aplicația `restlab` dezvoltată în laboratoarele anterioare, care expune un API REST pentru gestionarea produselor. Structura proiectului, rezumată, este următoarea:

```
src/main/java/ro/unitbv/restlab
├── RestLabApplication.java
├── controller
│   └── ProductController.java
├── service
│   └── ProductService.java
├── repository
│   └── ProductRepository.java
├── model
│   └── Product.java
├── mapper
│   └── ProductMapper.java
├── dto
│   ├── request
│   │   ├── CreateProductRequest.java
│   │   └── UpdateProductRequest.java
│   └── response
│       ├── ProductResponse.java
│       ├── ErrorResponse.java
│       └── ValidationExceptionResponse.java
└── exception
    ├── ProductNotFoundException.java
    └── GlobalExceptionHandler.java
```

Endpoint-urile expuse de `ProductController` sunt:

- `GET /api/products` – listează toate produsele;
- `GET /api/products/{id}` – citește un produs după id;
- `POST /api/products` – creează un produs;
- `PUT /api/products/{id}` – actualizează un produs;
- `DELETE /api/products/{id}` – șterge un produs.

Asupra acestei aplicații vor fi aplicate, pe rând, mai multe scheme de autentificare și autorizare. Pe parcursul laboratorului se va reveni, în mod repetat, la acest controller, pentru a vedea cum se modifică comportamentul vizibil la client.

3.9. Adăugarea dependenței Spring Security

Prima modificare constă în adăugarea, în `pom.xml`, a dependenței `spring-boot-starter-security`:

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

Pentru testele de securitate se adaugă și:

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
    <scope>test</scope>
</dependency>
```

3.9.1. Ce se întâmplă imediat după adăugarea dependenței

Mecanismul de autoconfigurare al Spring Boot detectează prezența Spring Security pe classpath[18] și activează, automat, o configurație implicită. Fără nicio linie de cod suplimentară, efectele vizibile sunt:

- toate endpoint-urile devin protejate – orice cerere primește răspunsul `401 Unauthorized` dacă nu conține credențiale valide;
- este activată autentificarea **HTTP Basic** și **form login**;
- este generat un utilizator implicit, cu username-ul `user` și o parolă aleatorie afișată în consolă la pornire, sub forma:

  ```
  Using generated security password: 8a7f3b2c-4e1d-4f5a-9c8b-12a3b4c5d6e7
  ```

- este activată protecția CSRF[19] (relevantă pentru formulare, dar problematică pentru API-urile REST).

Observație practică: Acest comportament implicit este util pentru a demonstra că Spring Security este activ, dar nu este potrivit pentru producție. În secțiunile următoare se va vedea cum se înlocuiește cu o configurație explicită. [3]

3.10. Testarea comportamentului implicit

Se pornește aplicația și se execută o cerere fără autentificare:

[18] Classpath este lista de locații (directoare și arhive JAR) în care mașina virtuală Java caută clasele și resursele necesare la rularea unei aplicații.
[19] CSRF (Cross-Site Request Forgery) este un atac prin care un site malițios determină browserul unui utilizator autentificat să trimită cereri neautorizate către o altă aplicație în numele acestuia.

```
curl -i http://localhost:8080/api/products
```

Răspunsul primit este:

```
HTTP/1.1 401
WWW-Authenticate: Basic realm="Realm"
Content-Type: application/json

{"status":401,"error":"Unauthorized","path":"/api/products"}
```

Prezența antetului `WWW-Authenticate: Basic` semnalează clientului că serverul așteaptă autentificare de tip HTTP Basic. Cu credențialele implicite, cererea devine:

```
curl -i -u user:8a7f3b2c-4e1d-4f5a-9c8b-12a3b4c5d6e7 \
     http://localhost:8080/api/products
```

și întoarce lista produselor, cu statusul `200 OK`.

3.11. Prima configurație explicită: SecurityFilterChain

Începând cu Spring Security 5.7 (și cu Spring Boot 3.x), clasa `WebSecurityConfigurerAdapter` a fost eliminată. Configurația se face acum prin declararea unor bean-uri, modelul fiind mai explicit și mai testabil. [3, 24]

Se creează pachetul `ro.unitbv.restlab.config` și în el clasa `SecurityConfig`:

```
package ro.unitbv.restlab.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
                                                        throws Exception {
        http.authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated()
            )
            .httpBasic(basic -> {});

        return http.build();
    }
}
```

Această configurație minimă declară:

- orice cerere trebuie să fie autentificată (`anyRequest().authenticated()`);
- este activat mecanismul HTTP Basic (`httpBasic(basic -> {})`).

Prezența bean-ului `SecurityFilterChain` dezactivează autoconfigurarea implicită, astfel încât se obține control complet asupra regulilor de securitate.

3.11.1. Anatomia DSL-ului HttpSecurity

Obiectul `HttpSecurity` este un *builder* fluent, structurat pe grupuri logice de configurații. Fiecare metodă primește un *lambda* (`Customizer`) care permite configurarea unui subsistem:

- `authorizeHttpRequests` – reguli de autorizare pe URL;
- `httpBasic` – autentificare cu antetul Authorization: Basic;
- `formLogin` – autentificare prin formular HTML;
- `sessionManagement` – strategia de gestionare a sesiunii;
- `csrf` – protecția împotriva atacurilor CSRF;
- `cors` – configurația Cross-Origin Resource Sharing;
- `exceptionHandling` – tratarea erorilor de autentificare și autorizare;
- `oauth2ResourceServer` – configurație pentru server de resurse OAuth2/JWT.

Stilul recomandat este *lambda DSL*[20], în care fiecare grup de configurații este exprimat printr-o expresie lambda. Stilul vechi, cu `.and()`, este marcat ca depreciat.

3.12. Autentificare HTTP Basic

HTTP Basic este cel mai simplu mecanism de autentificare specificat de RFC 7617. Credențialele sunt trimise într-un antet HTTP sub forma:

```
Authorization: Basic <Base64(username:password)>
```

Codificarea Base64 nu este criptare – este o transformare reversibilă care doar asigură că șirul poate fi transportat într-un antet HTTP. În lipsa unui canal TLS (HTTPS), credențialele sunt, practic, transmise în clar. Din acest motiv, HTTP Basic este acceptabil numai peste HTTPS. [3, 24]

3.12.1. Cum procesează Spring Security un antet Basic

Prezența configurației `httpBasic(...)` adaugă în lanț filtrul `BasicAuthenticationFilter`. Pentru fiecare cerere, acesta:

1. Citește antetul `Authorization` și verifică dacă începe cu `Basic ` ;
2. Decodifică partea Base64 și o împarte în `username` și `password`;
3. Construiește un `UsernamePasswordAuthenticationToken` neautentificat;
4. Îl trimite către `AuthenticationManager`;
5. În caz de succes, plasează rezultatul în SecurityContext și lasă cererea să continue;

[20] Un DSL este un limbaj specializat pentru un domeniu restrâns, iar un lambda DSL este o variantă a acestuia construită folosind expresii lambda pentru a permite configurări fluente și concise.

6. În caz de eșec, invocă `BasicAuthenticationEntryPoint`, care produce un răspuns `401` cu antetul `WWW-Authenticate`.

3.12.2. Avantaje și limitări

Avantajele HTTP Basic:

- este foarte simplu de implementat, atât pe server cât și pe client;
- nu are stare – credențialele se trimit la fiecare cerere;
- funcționează în orice librărie HTTP, fără biblioteci suplimentare.

Limitările:

- credențialele sunt trimise la fiecare cerere, ceea ce crește suprafața de atac;
- nu există mecanism nativ de logout – clientul trebuie să uite credențialele singur;
- nu se pretează la utilizatori expirați, blocați sau cu token-uri revocabile fără intervenție la bază de date la fiecare cerere.

În practică, HTTP Basic este potrivit pentru integrări între servicii interne sau pentru teste rapide. Pentru producție, variante mai potrivite sunt JWT sau OAuth2.

3.13. Autentificare cu formular (Form Login)

Form Login este mecanismul clasic folosit de aplicațiile web tradiționale: utilizatorul introduce numele și parola într-un formular HTML, iar serverul creează o sesiune ce rămâne validă până la logout. [3]

Chiar dacă aplicația curentă este un API REST, este util ca mecanismul să fie înțeles, pentru că apare frecvent în aplicațiile mixte (backend cu pagini de admin și API pentru frontend).

3.13.1. Configurație minimă

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/login", "/css/**").permitAll()
            .anyRequest().authenticated()
        )
        .formLogin(form -> form
            .loginPage("/login").permitAll()
            .defaultSuccessUrl("/home", true)
            .failureUrl("/login?error")
        )
        .logout(logout -> logout
            .logoutUrl("/logout")
            .logoutSuccessUrl("/login?logout")
        );

    return http.build();
}
```

Explicații punctuale:

- `loginPage("/login")` – indică faptul că pagina de login va fi servită de aplicație (nu cea implicită). Fără acest apel, Spring Security servește automat o pagină HTML simplă, utilă pentru teste;
- `defaultSuccessUrl("/home", true)` – URL-ul la care se redirecționează utilizatorul după autentificare reușită; al doilea argument `true` forțează redirectarea către `/home` chiar dacă utilizatorul încercase inițial alt URL;
- `failureUrl("/login?error")` – URL-ul afișat la eșecul autentificării;
- `logout(...)` – activează filtrul de logout, care, la o cerere `POST /logout`, invalidează sesiunea.

3.13.2. Handler-e de succes și eșec

Pentru scenarii mai complexe (de exemplu, pentru a returna JSON în loc de a redirecta), se pot folosi handler-e dedicate:

```
.formLogin(form -> form
    .successHandler((request, response, authentication) -> {
        response.setStatus(HttpStatus.OK.value());
        response.getWriter().write("{\"status\":\"ok\"}");
    })
    .failureHandler((request, response, exception) -> {
        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.getWriter().write("{\"error\":\"bad credentials\"}");
    })
)
```

Aceste handler-e se folosesc rar într-un API pur REST, dar este util să se știe că punctele de extensie există.

3.14. Definirea utilizatorilor în memorie

Utilizatorul implicit generat de Spring Boot este util doar pentru verificare rapidă. Într-o aplicație reală, utilizatorii trebuie definiți explicit. Prima variantă, cea mai simplă, este definirea lor în memorie, direct în configurație. [3]

3.14.1. Exemplu cu InMemoryUserDetailsManager

```
package ro.unitbv.restlab.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.provisioning.InMemoryUserDetailsManager;

@Configuration
public class UserConfig {
```

```
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public InMemoryUserDetailsManager userDetailsService(
                                                PasswordEncoder encoder) {
        UserDetails user = User.builder()
                .username("user")
                .password(encoder.encode("user123"))
                .roles("USER")
                .build();

        UserDetails admin = User.builder()
                .username("admin")
                .password(encoder.encode("admin123"))
                .roles("ADMIN", "USER")
                .build();

        return new InMemoryUserDetailsManager(user, admin);
    }
}
```

Observații:

- `User.builder()` construiește un `UserDetails` – reprezentarea utilizatorului în Spring Security;
- `roles("ADMIN", "USER")` generează automat autoritățile `ROLE_ADMIN` și `ROLE_USER` (prefixul este adăugat implicit);
- `InMemoryUserDetailsManager` este o implementare `UserDetailsService` care ține utilizatorii într-o structură `Map`;
- bean-ul de tip `PasswordEncoder` este declarat separat, astfel încât să poată fi reutilizat și de alte componente.

3.14.2. Prefixul ROLE_

În Spring Security, un **rol** este, formal, o *authority* al cărei nume începe cu `ROLE_`. Metodele `hasRole("ADMIN")` și `hasAuthority("ROLE_ADMIN")` sunt echivalente; prima adaugă prefixul automat, a doua îl așteaptă scris complet.

Confundarea celor două cauzează erori frecvente. Regula mnemonică:

- `hasRole(X)` ⇔ permisiune numită `ROLE_X`;
- `hasAuthority(X)` ⇔ permisiune numită exact `X`.

3.14.3. PasswordEncoder: de ce nu se stochează parolele în clar

Stocarea parolelor în clar într-o bază de date este una dintre cele mai grave erori de securitate. În cazul unei scurgeri de date, toate parolele devin accesibile. Chiar și folosirea

unor funcții hash simple (MD5, SHA-1) este insuficientă: tabelele precalculate (rainbow tables) permit recuperarea rapidă a parolelor uzuale.

Soluția consacrată este folosirea unor funcții de codificare proiectate special pentru parole – lente, cu factor de cost configurabil și cu salt aleator automat. Spring Security oferă o interfață `PasswordEncoder` cu metodele:

```
public interface PasswordEncoder {
    String encode(CharSequence rawPassword);
    boolean matches(CharSequence rawPassword, String encodedPassword);
    default boolean upgradeEncoding(String encodedPassword){
      return false; }
}
```

3.14.4. BCrypt

`BCryptPasswordEncoder` este implementarea recomandată pentru aplicații noi. Caracteristici:

- folosește un algoritm bazat pe Blowfish, proiectat pentru a fi lent în mod intenționat;
- generează un **salt aleator** pentru fiecare parolă, inclus direct în șirul rezultat – nu este nevoie să fie stocat separat;
- permite configurarea unui factor de cost (work factor) între 4 și 31; valoarea implicită este 10, ceea ce înseamnă 2^{10} iterații interne.

Un exemplu de parolă codificată cu BCrypt arată astfel:

```
$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
```

Structura este:

- `$2a$` – identificatorul algoritmului;
- `$10$` – factorul de cost;
- următoarele 22 de caractere – salt-ul codificat;
- restul – hash-ul propriu-zis.

Pentru a compara o parolă introdusă de utilizator cu una stocată, BCrypt extrage salt-ul și costul din șirul stocat, aplică din nou algoritmul asupra parolei introduse și compară rezultatul. **Nu există decriptare** – parola originală nu poate fi recuperată.

3.14.5. Alegerea factorului de cost

Factorul de cost trebuie ales astfel încât o singură verificare să dureze în jur de 100–300 ms pe hardware-ul serverului. Valoarea implicită (10) este rezonabilă pentru majoritatea aplicațiilor. O valoare prea mică face parolele vulnerabile la atacuri brute-force; o valoare prea mare poate bloca serverul sub încărcare.

3.14.6. DelegatingPasswordEncoder

Spring Security oferă și `DelegatingPasswordEncoder` – un `PasswordEncoder` care alege algoritmul în funcție de un prefix scris în parola stocată:

```
{bcrypt}$2a$10$N9qo8uLOick...
{noop}user123
{pbkdf2}5d923b44a6...
```

Acest mecanism permite coexistența în bază de date a parolelor codificate cu algoritmi diferiți și migrarea treptată către algoritmi mai puternici. Se obține astfel:

```
@Bean
public PasswordEncoder passwordEncoder() {
    return PasswordEncoderFactories.createDelegatingPasswordEncoder();
}
```

3.15. Utilizatori persistenți: UserDetailsService cu JPA

Varianta in-memory este utilă pentru teste, dar într-o aplicație reală utilizatorii trebuie să provină din baza de date. Pentru aceasta se introduce o entitate nouă, un repository și o implementare proprie a `UserDetailsService`. [3, 5]

3.15.1. Entitatea AppUser

```
package ro.unitbv.restlab.model;

import jakarta.persistence.*;
import lombok.*;

import java.util.HashSet;
import java.util.Set;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Builder
@Entity
@Table(name = "app_users")
public class AppUser {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, unique = true, length = 100)
    private String username;

    @Column(nullable = false)
    private String password;
```

```
    @Column(nullable = false)
    private boolean enabled;

    @ElementCollection(fetch = FetchType.EAGER)
    @CollectionTable(
        name = "app_user_roles",
        joinColumns = @JoinColumn(name = "user_id")
    )
    @Column(name = "role")
    @Builder.Default
    private Set<String> roles = new HashSet<>();
}
```

Comentarii:

- numele tabelei este `app_users` – cuvântul `users` este rezervat în unele baze de date (PostgreSQL, Oracle);
- `@ElementCollection` generează un tabel suplimentar `app_user_roles` cu perechi (user_id, role); modelul este minim, dar suficient pentru majoritatea aplicațiilor;
- câmpul `enabled` permite dezactivarea conturilor fără a le șterge;
- rolurile sunt stocate ca șiruri – aici, prin convenție, se vor folosi valori fără prefix (`ADMIN`, `USER`), iar prefixul `ROLE_` va fi adăugat explicit la construirea `UserDetails`.

3.16. Repository-ul AppUserRepository

```
package ro.unitbv.restlab.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import ro.unitbv.restlab.model.AppUser;

import java.util.Optional;

public interface AppUserRepository extends JpaRepository<AppUser, Long> {

    Optional<AppUser> findByUsername(String username);

    boolean existsByUsername(String username);
}
```

Metoda `findByUsername` este derivată automat din numele metodei; Spring Data JPA generează, în spate, interogarea SQL corespunzătoare.

3.16.1. Implementarea UserDetailsService

```
package ro.unitbv.restlab.security;

import lombok.RequiredArgsConstructor;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import
org.springframework.security.core.userdetails.UsernameNotFoundException;
```

```
import org.springframework.stereotype.Service;
import ro.unitbv.restlab.repository.AppUserRepository;

import java.util.Set;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
public class AppUserDetailsService implements UserDetailsService {

    private final AppUserRepository userRepository;

    @Override
    public UserDetails loadUserByUsername(String username) {
        return userRepository.findByUsername(username)
            .map(u -> User.builder()
                .username(u.getUsername())
                .password(u.getPassword())
                .disabled(!u.isEnabled())
                .authorities(mapAuthorities(u.getRoles()))
                .build())
            .orElseThrow(() ->
                new UsernameNotFoundException(
                    "Utilizatorul " + username + " nu există."));
    }

    private Set<SimpleGrantedAuthority> mapAuthorities(Set<String> roles) {
        return roles.stream()
            .map(r -> new SimpleGrantedAuthority("ROLE_" + r))
            .collect(Collectors.toSet());
    }
}
```

Componenta aceasta este suficientă pentru ca `DaoAuthenticationProvider` să poată valida automat utilizatorii proveniți din baza de date. Spring Security detectează, prin autoconfigurare, prezența unui singur bean de tip `UserDetailsService` și îl folosește ca sursă de utilizatori.

3.16.2. Popularea inițială a utilizatorilor

Pentru ca aplicația să aibă utilizatori la prima pornire, se poate folosi un `CommandLineRunner`:

```
package ro.unitbv.restlab.config;

import lombok.RequiredArgsConstructor;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.password.PasswordEncoder;
import ro.unitbv.restlab.model.AppUser;
import ro.unitbv.restlab.repository.AppUserRepository;

import java.util.Set;
```

```
@Configuration
@RequiredArgsConstructor
public class DataSeeder {

    @Bean
    public CommandLineRunner seedUsers(AppUserRepository repo,
                                       PasswordEncoder encoder) {
        return args -> {
            if (!repo.existsByUsername("admin")) {
                repo.save(AppUser.builder()
                    .username("admin")
                    .password(encoder.encode("admin123"))
                    .enabled(true)
                    .roles(Set.of("ADMIN", "USER"))
                    .build());
            }
            if (!repo.existsByUsername("user")) {
                repo.save(AppUser.builder()
                    .username("user")
                    .password(encoder.encode("user123"))
                    .enabled(true)
                    .roles(Set.of("USER"))
                    .build());
            }
        };
    }
}
```

La fiecare pornire, se verifică dacă utilizatorii există. Dacă nu, se creează. Parolele sunt codificate înainte de a fi salvate – **niciodată** nu se stochează parole în clar.

3.17. Autorizare la nivel de URL

Prima formă de autorizare este cea **declarativă**, prin reguli aplicate în `HttpSecurity`. Aici se definește, pentru fiecare tipar de URL, cine are voie să acceseze. [3, 24]

3.17.1. Exemplu aplicat pe ProductController

Se presupune că operațiile de citire (`GET`) trebuie să fie permise oricărui utilizator autentificat, iar operațiile de scriere (`POST`, `PUT`, `DELETE`) numai administratorilor:

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    http
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(HttpMethod.GET,"/api/products/**")
            .hasAnyRole("USER", "ADMIN")
            .requestMatchers(HttpMethod.POST,"/api/products/**")
            .hasRole("ADMIN")
            .requestMatchers(HttpMethod.PUT, "/api/products/**")
            .hasRole("ADMIN")
            .requestMatchers(HttpMethod.DELETE, "/api/products/**")
            .hasRole("ADMIN")
```

```
                .requestMatchers("/h2-console/**").permitAll()
                .anyRequest().authenticated()
            )
            .httpBasic(basic -> {})
            .csrf(csrf -> csrf.disable())
            .headers(h -> h.frameOptions(f -> f.disable()));

        return http.build();
    }
```

Elemente de remarcat:

- `requestMatchers(HttpMethod.GET, "/api/products/**")` – matcher specific pentru metoda HTTP și un pattern de URL;
- regulile sunt evaluate **în ordinea declarării** – prima regulă care se potrivește determină deciziile; `anyRequest()` trebuie să rămână ultima;
- `permitAll()` – permite accesul fără autentificare (util pentru `/h2-console` în dezvoltare);
- `headers(h -> h.frameOptions(f -> f.disable()))` – necesar pentru ca H2 console (afișat într-un iframe) să funcționeze;
- `csrf(csrf -> csrf.disable())` – dezactivarea protecției CSRF.

3.17.2. Matchere disponibile

Spring Security oferă mai multe tipuri de matchere pentru URL-uri:

- `requestMatchers(String... patterns)` – pattern-uri Ant (cu wildcards: `*`, `**`, `?`);
- `requestMatchers(HttpMethod method, String... patterns)` – filtrare și după metoda HTTP;
- `requestMatchers(RequestMatcher... matchers)` – matchere custom;
- `regexMatchers(String... regex)` – pattern-uri pe bază de expresii regulate.

Exemple de pattern-uri Ant:

- `/api/products` – exact acest URL;
- `/api/products/*` – orice URL care urmează după `/api/products/`, **fără** a include separatori ulteriori de cale;
- `/api/products/**` – orice URL care începe cu `/api/products/`, inclusiv subcăi.

3.17.3. hasRole, hasAuthority, hasAnyRole

Expresiile de autorizare cele mai utilizate:

- `permitAll()` – acces liber;
- `denyAll()` – blocare totală;
- `authenticated()` – orice utilizator autentificat;
- `hasRole("ADMIN")` – utilizator cu autoritatea `ROLE_ADMIN`;
- `hasAnyRole("ADMIN", "USER")` – cel puțin unul dintre rolurile listate;
- `hasAuthority("READ_PRODUCTS")` – autoritate cu numele exact indicat;

- `access("expression")` – expresie SpEL[21] generală.

3.18. Autorizare la nivel de metodă (Method Security)

Regulile din `HttpSecurity` sunt clare, dar sunt legate strâns de URL. Pentru decizii mai fine, de exemplu „un utilizator poate modifica numai produsele create de el”, se folosește **Method Security** – adnotări aplicate direct pe metodele din service sau controller. [3, 24]

3.18.1. Activarea Method Security

```
@Configuration
@EnableMethodSecurity
public class MethodSecurityConfig {
}
```

`@EnableMethodSecurity` activează, implicit, procesarea adnotărilor `@PreAuthorize` și `@PostAuthorize`. Pentru adnotarea mai veche `@Secured`, trebuie activat explicit:

```
@EnableMethodSecurity(securedEnabled = true)
```

Pentru `@RolesAllowed` (din JSR-250):

```
@EnableMethodSecurity(jsr250Enabled = true)
```

3.18.2. @PreAuthorize

`@PreAuthorize` evaluează o expresie SpEL **înainte** de a executa metoda. Dacă expresia este falsă, metoda nu este apelată și se aruncă `AccessDeniedException`.

```
@Service
@RequiredArgsConstructor
public class ProductService {

    private final ProductRepository productRepository;
    private final ProductMapper productMapper;

    @PreAuthorize("hasRole('USER') or hasRole('ADMIN')")
    public List<ProductResponse> findAll() { /* ... */ }

    @PreAuthorize("hasRole('ADMIN')")
    public ProductResponse create(CreateProductRequest request) { /*...*/ }

    @PreAuthorize("hasRole('ADMIN') and #id > 0")
    public void deleteById(Integer id) { /* ... */ }
}
```

În expresii se pot referi **argumentele metodei** prin sintaxa `#numeArgument` și obiectul `principal` prin variabilele predefinite `authentication` sau `principal`.

[21] SpEL (Spring Expression Language) este un limbaj de expresii din Spring Framework folosit pentru a evalua dinamic valori, a accesa proprietăți și a defini reguli în configurații sau cod.

3.18.3. @PostAuthorize

`@PostAuthorize` evaluează o expresie **după** execuția metodei, cu acces la rezultatul returnat prin variabila `returnObject`:

```
@PostAuthorize("returnObject.ownerUsername == authentication.name or
hasRole('ADMIN')")
public ProductResponse findById(Integer id) { /* ... */ }
```

Se folosește atunci când decizia depinde de conținutul rezultatului – de exemplu, un utilizator poate citi doar resursele al căror proprietar este el însuși.

Atenție: `@PostAuthorize` execută efectiv metoda și apoi verifică. Dacă metoda are efecte secundare (de exemplu creează ceva), ele se produc chiar dacă autorizarea eșuează. Pentru operații de modificare se preferă `@PreAuthorize`.

3.18.4. @Secured și @RolesAllowed

Ambele adnotări acceptă o listă de roluri, dar cu sintaxe diferite:

```
@Secured({"ROLE_ADMIN", "ROLE_MANAGER"})
public void doSomething() { /* ... */ }

@RolesAllowed({"ADMIN", "MANAGER"})
public void doSomethingElse() { /* ... */ }
```

`@Secured` așteaptă numele complete ale autorităților (cu prefixul `ROLE_`), iar `@RolesAllowed` adaugă automat prefixul. Ambele sunt inferioare, ca expresivitate, lui `@PreAuthorize`, care suportă expresii SpEL complexe.

3.18.5. Cum funcționează Method Security intern

`@EnableMethodSecurity` declanșează crearea unor **proxy-uri AOP**[22] în jurul bean-urilor Spring care conțin metode adnotate. Proxy-ul interceptează apelul, evaluează expresia, și fie lasă apelul să treacă, fie aruncă `AccessDeniedException`.

Consecințe practice:

- adnotările funcționează numai pe **bean-uri Spring** (clase marcate cu `@Service`, `@Component` etc.);
- apelurile interne (dintr-o metodă a aceleiași clase către alta) **nu trec prin proxy** și, deci, nu sunt verificate; dacă se dorește acest lucru, metodele trebuie apelate prin bean-ul Spring (de exemplu prin auto-injectare).

[22] AOP (Aspect-Oriented Programming) este o paradigmă care separă preocupările transversale (ex. logging, securitate) de logica de business, aplicându-le modular asupra codului. Un proxy AOP este un obiect intermediar generat care interceptează apelurile către un obiect țintă pentru a aplica automat logica definită în aspecte (ex. logging, securitate).

4. CSRF – Cross-Site Request Forgery

CSRF (Cross-Site Request Forgery) este un tip de atac în care un site rău-intenționat folosește sesiunea autentificată a unui utilizator (stocată într-un cookie) pentru a executa acțiuni în numele lui, fără consimțământ. [3, 24]

4.1. Scenariul atacului

1. Un utilizator se autentifică la banca sa online, iar browserul primește un cookie de sesiune.
2. Fără a se deautentifica, același utilizator vizitează un site rău-intenționat.
3. Site-ul include, în mod ascuns, un `<form>` sau o imagine `<img src="...">` care trimite o cerere către banca.
4. Browserul, respectând regula de trimitere a cookie-urilor pentru domeniul băncii, atașează cookie-ul de sesiune la cerere.
5. Serverul băncii primește cererea, o consideră legitimă și execută acțiunea (de exemplu, un transfer).

4.2. Mecanismul de apărare

Spring Security activează, implicit, un filtru care:

- generează la prima cerere un token CSRF aleator;
- îl atașează sesiunii și îl trimite clientului (prin `_csrf` în formulare sau prin antetul `X-XSRF-TOKEN`);
- la fiecare cerere de modificare (POST, PUT, DELETE, PATCH), așteaptă ca tokenul să fie retransmis.

Pentru că un atacator nu poate citi cookie-urile sau conținutul paginilor victimei (datorită politicii **same-origin**), el nu poate afla tokenul și, deci, nu poate efectua cereri valide.

4.3. De ce se dezactivează CSRF într-un API stateless

Atacurile CSRF exploatează cookie-urile de sesiune. Un API REST care:

- nu folosește cookie-uri, ci antete de tip `Authorization: Bearer ...`;
- este stateless (nu păstrează sesiune pe server);

este imun, prin construcție, la CSRF. În acest caz, protecția este inutilă și poate fi dezactivată fără a crea vulnerabilități:

```
.csrf(csrf -> csrf.disable())
```

> **Regulă practică**
>
> CSRF se dezactivează numai pentru API-uri pur stateless care folosesc token-uri în antet. Dacă aplicația are formulare HTML sau folosește cookie-uri de sesiune pentru autentificare, CSRF trebuie activ.

4.4. Varianta cu cookie (pentru SPA)

Dacă aplicația este un Single Page Application (SPA) care consumă API-ul și folosește cookie-uri, se preferă `CookieCsrfTokenRepository.withHttpOnlyFalse()`. În acest caz, tokenul este plasat într-un cookie înțeles de JavaScript, iar clientul îl citește și îl trimite înapoi în antetul `X-XSRF-TOKEN`:

```
.csrf(csrf -> csrf
    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
)
```

5. Managementul sesiunilor

O sesiune HTTP este un mecanism prin care serverul păstrează, între cereri succesive ale aceluiași client, o stare asociată acestuia. Clientul este identificat printr-un cookie (implicit `JSESSIONID`), iar serverul menține un obiect de sesiune în memorie sau într-un store partajat.

5.1. SessionCreationPolicy

Spring Security oferă patru politici, exprimate prin enumerarea `SessionCreationPolicy`:

- `ALWAYS` – se creează sesiune chiar dacă nu este necesară;
- `IF_REQUIRED` – valoarea implicită; se creează sesiune doar dacă este nevoie;
- `NEVER` – nu se creează sesiuni noi, dar se folosește cea existentă (dacă există);
- `STATELESS` – niciodată nu se folosesc sesiuni – fiecare cerere este autonomă.

5.2. Configurarea modului stateless

Pentru un API REST care folosește JWT, configurația corectă este:

```
.sessionManagement(sm -> sm
    .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
)
```

Efectele acestei configurații:

- nu se creează cookie-ul `JSESSIONID`;
- SecurityContextHolder este resetat la finalul fiecărei cereri;
- fiecare cerere trebuie să conțină propriile credențiale (token);
- protecția CSRF bazată pe sesiune devine, în mod natural, inutilă.

5.3. Stateful vs stateless

Alegerea dintre cele două modele nu este doar tehnică, ci și arhitecturală:

- **Stateful** – simplu de implementat pentru aplicații monolitice, pagini server-side, dar dificil de scalat orizontal (se cere sesiune partajată între instanțe) și expus la atacuri CSRF;

- **Stateless** – potrivit pentru API-uri consumate de SPA sau aplicații mobile, scalabil, dar presupune gestionarea corectă a token-urilor (expirare, revocare, reîmprospătare).

6. Autentificare cu JWT (JSON Web Tokens)

JWT este un standard (RFC 7519) pentru transportul de **claim-uri** (afirmații) într-un format compact, auto-descriptiv și semnat criptografic. În contextul autentificării, un JWT este folosit ca token de acces care conține, în mod sigur, identitatea și permisiunile utilizatorului.

6.1. Structura unui JWT

Un JWT este compus din trei părți separate prin punct:

```
<header>.<payload>.<signature>
```

Fiecare parte este codificată Base64URL. Un exemplu concret:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiJhZG1pbiIsInJvbGVzIjpbIlJPTEVfQURNSU4iXSwiaWF0IjoxNzAwMDAwMDAwLCJ
leHAiOjE3MDAwMDM2MDB9.
dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

Decodificate, cele trei părți conțin, în ordine:

6.1.1. Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Specifică algoritmul de semnare (`HS256` = HMAC SHA-256) și tipul tokenului. Alte valori uzuale pentru `alg` sunt `RS256` (semnare asimetrică RSA) sau `ES256` (ECDSA).

6.1.2. Payload

```
{
  "sub": "admin",
  "roles": ["ROLE_ADMIN"],
  "iat": 1700000000,
  "exp": 1700003600
}
```

Conține claim-uri standard definite de RFC:

- `sub` (subject) – identitatea subiectului (de regulă username-ul);
- `iat` (issued at) – momentul emiterii tokenului (epoch, secunde);
- `exp` (expiration) – momentul expirării;
- `iss` (issuer), `aud` (audience), `nbf` (not before) – claim-uri opționale standard;

- restul sunt claim-uri custom (de exemplu, lista rolurilor).

6.1.3. Signature

Semnătura se calculează aplicând algoritmul din header peste concatenarea `base64url(header) + "." + base64url(payload)` folosind o cheie secretă cunoscută doar serverului. Matematic:

```
HMACSHA256(base64url(header) + "." + base64url(payload), secret_key)
```

Important: Payload-ul **NU** este criptat – este doar codificat Base64, deci oricine poate citi conținutul unui JWT. Nu se pun niciodată informații sensibile (parole, numere de card) în payload. Ce asigură JWT este autenticitatea: orice modificare a payload-ului invalidează semnătura și tokenul devine respins.

6.2. Fluxul de autentificare cu JWT

1. Clientul trimite credențialele către `POST /api/auth/login`.
2. Serverul le validează (`UserDetailsService` + `PasswordEncoder`).
3. Dacă sunt corecte, generează un JWT cu identitatea și rolurile utilizatorului.
4. Clientul primește tokenul și îl stochează (în memoria aplicației, localStorage, etc.).
5. La fiecare cerere ulterioară, clientul adaugă antetul `Authorization: Bearer <token>`.
6. Un filtru pe server extrage tokenul, îi verifică semnătura, îi controlează expirarea și, dacă totul este în regulă, populează SecurityContext cu identitatea extrasă din payload.
7. Nu se păstrează sesiune pe server – fiecare cerere este autonomă.

6.3. Adăugarea bibliotecii JJWT

Se folosește biblioteca `jjwt` (Java JWT):

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.12.6</version>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>

<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.12.6</version>
    <scope>runtime</scope>
</dependency>
```

6.4. Configurarea cheii secrete

Cheia secretă folosită la semnare se configurează în `application.properties`:

```
security.jwt.secret=4f8a7b3c2d1e9f0a6b5c4d3e2f1a0b9c8d7e6f5a4b3c2d1e0f9a8b7c6d5e4f3a
security.jwt.expiration-ms=3600000
```

Observații:

- cheia trebuie să fie lungă (cel puțin 256 de biți = 32 de bytes = 64 de caractere hex) pentru `HS256`;
- în producție, cheia NU se pune în cod sau în fișiere de configurație trimise în repository; se folosesc variabile de mediu sau un vault;
- durata de expirare uzuală este 15–60 de minute; pentru durate lungi se folosesc separat **refresh tokens**.

6.5. Serviciul JwtService

```
package ro.unitbv.restlab.security;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;

import javax.crypto.SecretKey;
import java.util.Date;
import java.util.List;
import java.util.function.Function;
import java.util.stream.Collectors;

@Service
public class JwtService {

    @Value("${security.jwt.secret}")
    private String secret;

    @Value("${security.jwt.expiration-ms}")
    private long expirationMs;

    private SecretKey signingKey() {
        return Keys.hmacShaKeyFor(Decoders.BASE64.decode(secret));
    }

    public String generateToken(UserDetails userDetails) {
        List<String> roles = userDetails.getAuthorities().stream()
                .map(GrantedAuthority::getAuthority)
                .toList();

        Date now = new Date();
```

```
        Date expiration = new Date(now.getTime() + expirationMs);

        return Jwts.builder()
                .subject(userDetails.getUsername())
                .claim("roles", roles)
                .issuedAt(now)
                .expiration(expiration)
                .signWith(signingKey())
                .compact();
    }

    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    @SuppressWarnings("unchecked")
    public List<String> extractRoles(String token) {
        return extractClaim(token, c -> (List<String>) c.get("roles"));
    }

    public boolean isTokenValid(String token, UserDetails userDetails) {
        String username = extractUsername(token);
        return username.equals(
                        userDetails.getUsername()) && !isExpired(token);
    }

    private boolean isExpired(String token) {
        return extractClaim(token, Claims::getExpiration)
               .before(new Date());
    }

    private <T> T extractClaim(String token,
                               Function<Claims, T> resolver) {
        Claims claims = Jwts.parser()
                .verifyWith(signingKey())
                .build()
                .parseSignedClaims(token)
                .getPayload();
        return resolver.apply(claims);
    }
}
```

Comentarii:

- `Jwts.builder()` construiește un token nou, setând claim-urile și semnătura;
- `Jwts.parser().verifyWith(...)` verifică semnătura și, dacă este corectă, returnează claim-urile; dacă semnătura este invalidă, aruncă excepție;
- cheia este decodificată din Base64, nu folosită direct ca șir (pentru a permite stocarea ei în formate standard).

6.6. Filtrul JwtAuthenticationFilter

Filtrul personalizat este responsabil să extragă tokenul din antet, să îl valideze și să populeze contextul de securitate. Se extinde `OncePerRequestFilter`, care garantează execuția o singură dată per cerere [3]:

```
package ro.unitbv.restlab.security;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.security.authentication
                                    .UsernamePasswordAuthenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication
                                        .WebAuthenticationDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;
import java.io.IOException;

@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain filterChain)
            throws ServletException, IOException {

        String authHeader = request.getHeader("Authorization");

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            filterChain.doFilter(request, response);
            return;
        }

        String token = authHeader.substring(7);
        String username;

        try {
            username = jwtService.extractUsername(token);
        } catch (Exception e) {
            filterChain.doFilter(request, response);
            return;
        }
```

```
        if (username != null
            && SecurityContextHolder.getContext()
               .getAuthentication() == null) {

            UserDetails userDetails = userDetailsService
                                      .loadUserByUsername(username);

            if (jwtService.isTokenValid(token, userDetails)) {
                UsernamePasswordAuthenticationToken authToken =
                        new UsernamePasswordAuthenticationToken(
                                userDetails, null,
                                userDetails.getAuthorities());
                authToken.setDetails(
                        new WebAuthenticationDetailsSource()
                            .buildDetails(request));

               SecurityContextHolder.getContext()
                                    .setAuthentication(authToken);
            }
        }

        filterChain.doFilter(request, response);
    }
}
```

Elemente cheie de înțeles:

- filtrul **nu respinge** cererea dacă tokenul lipsește sau este invalid – o lasă să continue; respingerea, dacă este cazul, o face `AuthorizationFilter` mai târziu, pe baza regulilor declarate;
- se verifică dacă contextul nu a fost deja populat (evitare dublă autentificare);
- tokenul purtat în `UsernamePasswordAuthenticationToken` este cel care, populat în SecurityContext, va fi folosit de filtrele ulterioare.

6.7. Integrarea filtrului în SecurityFilterChain

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                    JwtAuthenticationFilter jwtFilter,
                                    AuthenticationEntryPoint entryPoint)
                                                    throws Exception {

    http
        .csrf(csrf -> csrf.disable())
        .cors(cors -> {})
        .sessionManagement(sm -> sm
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/auth/**").permitAll()
            .requestMatchers("/h2-console/**").permitAll()
            .requestMatchers(HttpMethod.GET, "/api/products/**")
            .hasAnyRole("USER", "ADMIN")
            .requestMatchers("/api/products/**").hasRole("ADMIN")
```

```
            .anyRequest().authenticated())
        .exceptionHandling(ex -> ex
            .authenticationEntryPoint(entryPoint))
        .addFilterBefore(jwtFilter,
                         UsernamePasswordAuthenticationFilter.class);
    return http.build();
}
```

Poziția critică este ultima linie: `addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class)`. Filtrul JWT trebuie să se execute **înainte** de filtrul clasic de autentificare cu parolă, astfel încât tokenul să fie procesat primul și să populeze contextul.

6.8. Endpoint-ul de login

Se adaugă un controller dedicat pentru operațiunile de autentificare:

```
package ro.unitbv.restlab.controller;

import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationT
oken;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.web.bind.annotation.*;
import ro.unitbv.restlab.dto.request.LoginRequest;
import ro.unitbv.restlab.dto.response.LoginResponse;
import ro.unitbv.restlab.security.JwtService;

@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthenticationManager authenticationManager;
    private final UserDetailsService userDetailsService;
    private final JwtService jwtService;

    @PostMapping("/login")
    public ResponseEntity<LoginResponse> login(
                                    @RequestBody LoginRequest request) {
        authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(
                request.getUsername(), request.getPassword()));

        UserDetails userDetails = userDetailsService.loadUserByUsername(
            request.getUsername());

        String token = jwtService.generateToken(userDetails);
```

```
            return ResponseEntity.ok(new LoginResponse(token));
        }
    }
```

DTO-urile sunt minime:

```
    @Getter
    @Setter
    @NoArgsConstructor
    @AllArgsConstructor
    public class LoginRequest {
        private String username;
        private String password;
    }

    @Getter
    @AllArgsConstructor
    public class LoginResponse {
        private String token;
    }
```

6.9. Publicarea AuthenticationManager ca bean

În Spring Security 6, `AuthenticationManager` nu mai este expus automat ca bean. Trebuie declarat explicit, astfel încât să poată fi injectat în controller:

```
    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }
```

6.10. Endpoint-ul de înregistrare

Pentru completitudine, se adaugă și un endpoint pentru crearea de utilizatori noi:

```
    @PostMapping("/register")
    public ResponseEntity<Void> register(
                                @Valid @RequestBody RegisterRequest request) {
        if (userRepository.existsByUsername(request.getUsername())) {
            return ResponseEntity.status(HttpStatus.CONFLICT).build();
        }

        AppUser user = AppUser.builder()
                .username(request.getUsername())
                .password(passwordEncoder.encode(request.getPassword()))
                .enabled(true)
                .roles(Set.of("USER"))
                .build();

        userRepository.save(user);
        return ResponseEntity.status(HttpStatus.CREATED).build();
    }
```

6.11. Testarea fluxului complet

Autentificare și obținerea tokenului:

```
curl -X POST http://localhost:8080/api/auth/login \
     -H "Content-Type: application/json" \
     -d '{"username":"admin","password":"admin123"}'
```

Răspunsul:

```
{"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
```

Folosirea tokenului la accesarea unui endpoint protejat:

```
curl -X GET http://localhost:8080/api/products \
     -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
```

Creare de produs (necesită rol ADMIN):

```
curl -X POST http://localhost:8080/api/products \
     -H "Authorization: Bearer <token>" \
     -H "Content-Type: application/json" \
     -d '{"name":"Laptop","price":3500.00,"stock":5}'
```

7. Varianta alternativă: OAuth2 Resource Server cu JWT

Implementarea JWT din secțiunea anterioară este didactică, dar necesită cod scris manual: filtru, serviciu de tokenuri, integrare în lanț. Spring Security oferă, de asemenea, o variantă **bazată pe standarde OAuth2**, care, odată configurată, face aceleași lucruri fără cod explicit. [3, 24]

7.1. Dependența necesară

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

7.2. Configurația

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws
Exception {
    http
        .csrf(csrf -> csrf.disable())
        .sessionManagement(sm -> sm
            .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        .authorizeHttpRequests(auth -> auth
            .requestMatchers(HttpMethod.GET, "/api/products/**")
            .hasAnyRole("USER", "ADMIN")
            .requestMatchers("/api/products/**").hasRole("ADMIN")
            .anyRequest().authenticated())
```

```
            .oauth2ResourceServer(oauth2 -> oauth2
                                    .jwt(jwt -> jwt.decoder(jwtDecoder())));

    return http.build();
}

@Bean
public JwtDecoder jwtDecoder() {
    SecretKey key = new SecretKeySpec(
        Decoders.BASE64.decode(secret), "HmacSHA256");
    return NimbusJwtDecoder.withSecretKey(key).build();
}
```

Ce oferă această variantă:

- filtrul `BearerTokenAuthenticationFilter` – extrage și validează automat tokenul;
- un convertor implicit de claim-uri în permisiuni (pentru un claim `scope` sau `scp`, prefixat cu `SCOPE_`);
- tratarea standard a erorilor (`401` pentru token invalid, `403` pentru permisiuni insuficiente).

7.3. Converter custom pentru claim-ul roles

Convertorul implicit mapează un claim `scope` la autorități de forma `SCOPE_X`. Pentru a păstra convenția `ROLE_X` folosită mai sus, se configurează un convertor propriu:

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter authoritiesConverter =
        new JwtGrantedAuthoritiesConverter();
    authoritiesConverter.setAuthoritiesClaimName("roles");
    authoritiesConverter.setAuthorityPrefix("");

    JwtAuthenticationConverter converter =
                                    new JwtAuthenticationConverter();
    converter.setJwtGrantedAuthoritiesConverter(authoritiesConverter);
    return converter;
}
```

Apoi, în lanț:

```
.oauth2ResourceServer(oauth2 -> oauth2
    .jwt(jwt -> jwt
        .decoder(jwtDecoder())
        .jwtAuthenticationConverter(jwtAuthenticationConverter())))
```

Varianta OAuth2 Resource Server se recomandă atunci când tokenurile sunt emise de un server de autorizare dedicat (Keycloak, Auth0, Okta). Pentru aplicații care emit propriile tokenuri, varianta cu filtru personalizat (custom) este adesea suficientă și mai ușor de înțeles la început.

8. CORS – Cross-Origin Resource Sharing

Browserele moderne aplică o politică de securitate numită same-origin policy: un script încărcat de pe un domeniu nu poate efectua, în mod implicit, cereri HTTP către alt domeniu. „Origin” este definit de tripletul (schemă, gazdă, port). [3, 17]

Într-o arhitectură tipică, un frontend SPA servit de pe un domeniu diferit de cel al API-ului are nevoie să poată face cereri către acesta. CORS este mecanismul prin care serverul poate spune explicit browserului: „accept cereri de la acest origin”.

8.1. Mecanismul CORS

Pentru cereri considerate „complexe” (orice altceva decât `GET`/`POST`/`HEAD` cu tipuri simple de conținut), browserul trimite mai întâi o cerere preflight de tip `OPTIONS`:

```
OPTIONS /api/products HTTP/1.1
Origin: https://frontend.example.com
Access-Control-Request-Method: POST
Access-Control-Request-Headers: Content-Type, Authorization
```

Dacă serverul răspunde corespunzător cu antete ca:

```
Access-Control-Allow-Origin: https://frontend.example.com
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type, Authorization
```

browserul trimite mai departe cererea reală. Altfel, o blochează. [17]

8.2. Configurarea CORS în Spring Security

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration configuration = new CorsConfiguration();

    configuration.setAllowedOrigins(List.of("https://fe.example.com"));
    configuration.setAllowedMethods(
                      List.of("GET", "POST", "PUT", "DELETE", "OPTIONS"));
    configuration.setAllowedHeaders(
                      List.of("Authorization", "Content-Type"));
    configuration.setAllowCredentials(true);
    configuration.setMaxAge(3600L);

    UrlBasedCorsConfigurationSource source =
                                    new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", configuration);
    return source;
}
```

Apoi, în `SecurityFilterChain`, se activează CORS:

```
.cors(cors -> {})
```

Spring Security detectează automat bean-ul `CorsConfigurationSource` și îl folosește. [17]

8.3. Greșeli frecvente în configurarea CORS

- folosirea `*` pentru `allowedOrigins` împreună cu `allowCredentials=true` – combinația este interzisă de specificație și browserele o resping;
- uitarea tratării preflight – dacă `OPTIONS` este blocat de `authorizeHttpRequests`, CORS nu funcționează; Spring Security, cu `cors()` activ, permite OPTIONS implicit;
- listarea doar domeniului, fără portul non-standard, atunci când frontendul rulează pe un port diferit.

9. Tratarea erorilor de securitate

În absența unei configurații explicite, Spring Security produce răspunsuri standard: `401 Unauthorized` la lipsă de autentificare, `403 Forbidden` la lipsă de permisiuni. Pentru un API care răspunde cu JSON este utilă personalizarea acestor răspunsuri, astfel încât clientul să primească un format uniform. [3]

9.1. AuthenticationEntryPoint – pentru 401

`AuthenticationEntryPoint` este invocat când o cerere neautentificată ajunge la un endpoint protejat. Varianta implicită trimite antetul `WWW-Authenticate` și un răspuns `401`. Se poate înlocui cu o componentă proprie:

```
package ro.unitbv.restlab.security;

import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.time.LocalDateTime;
import java.util.Map;

@Component
@RequiredArgsConstructor
public class RestAuthenticationEntryPoint implements
AuthenticationEntryPoint {

    private final ObjectMapper objectMapper;

    @Override
    public void commence(HttpServletRequest request,
                         HttpServletResponse response,
                         AuthenticationException authException)
```

```
            throws IOException {

        response.setStatus(HttpStatus.UNAUTHORIZED.value());
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);

        Map<String, Object> body = Map.of(
            "timestamp", LocalDateTime.now().toString(),
            "status", 401,
            "error", "Unauthorized",
            "message", authException.getMessage(),
            "path", request.getRequestURI()
        );

        objectMapper.writeValue(response.getOutputStream(), body);
    }
}
```

9.2. AccessDeniedHandler – pentru 403

`AccessDeniedHandler` este invocat atunci când utilizatorul este autentificat, dar nu are permisiunile necesare:

```
@Component
@RequiredArgsConstructor
public class RestAccessDeniedHandler implements AccessDeniedHandler {

    private final ObjectMapper objectMapper;

    @Override
    public void handle(HttpServletRequest request,
                       HttpServletResponse response,
                       AccessDeniedException accessDeniedException)
            throws IOException {

        response.setStatus(HttpStatus.FORBIDDEN.value());
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);


        Map<String, Object> body = Map.of(
            "timestamp", LocalDateTime.now().toString(),
            "status", 403,
            "error", "Forbidden",
            "message", accessDeniedException.getMessage(),
            "path", request.getRequestURI()
        );

        objectMapper.writeValue(response.getOutputStream(), body);
    }
}
```

9.3. Înregistrarea handler-elor

```
.exceptionHandling(ex -> ex
    .authenticationEntryPoint(restAuthenticationEntryPoint)
```

```
        .accessDeniedHandler(restAccessDeniedHandler))
```

Așa cum s-a discutat în secțiunea 3, filtrul `ExceptionTranslationFilter` este cel care decide care handler să fie apelat, în funcție de tipul excepției ridicate mai jos în lanț.

10. Plasarea regulilor de autorizare

O decizie importantă de arhitectură este unde se scriu regulile de autorizare. Există trei locuri posibile, fiecare cu avantaje și dezavantaje:

- **În SecurityFilterChain (declarativ, pe URL)**
  Avantaje: vizibilitate centralizată, nu „poluează" codul de business.
  Dezavantaje: reguli exprimate doar la nivel de URL; nu pot depinde de date.
- **Pe service (cu @PreAuthorize)**
  Avantaje: regulile sunt aproape de logica protejată; pot folosi argumentele metodei.
  Dezavantaje: regulile sunt împrăștiate; mai dificil de avut o imagine globală.
- **În controller (mixt)**
  Avantaje: regulile sunt aproape de endpoint-ul expus.
  Dezavantaje: dublează informația de mapare URL deja existentă.

Recomandare practică: Regulile de granularitate mare (pe grupuri de URL-uri) se scriu în `SecurityFilterChain`. Regulile fine, care depind de argumente sau de rezultatul metodei, se scriu cu `@PreAuthorize` sau `@PostAuthorize` pe service. Cele două abordări sunt complementare, nu alternative. [3, 24]

11. Aplicarea regulilor pe aplicația restlab – exemplu consolidat

În continuare se prezintă, într-o formă consolidată, modul în care laboratorul curent modifică aplicația existentă. Structura fișierelor după adăugarea securității:

```
src/main/java/ro/unitbv/restlab
├── RestLabApplication.java
├── config
│   ├── SecurityConfig.java          // (NOU)
│   └── DataSeeder.java              // (NOU)
├── controller
│   ├── ProductController.java       // (modificat)
│   └── AuthController.java          // (NOU)
├── service
│   └── ProductService.java          // (modificat - adnotări)
├── repository
│   ├── ProductRepository.java
│   └── AppUserRepository.java       // (NOU)
├── model
│   ├── Product.java
│   └── AppUser.java                 // (NOU)
├── mapper
│   └── ProductMapper.java
├── dto
│   ├── request
│   │   ├── CreateProductRequest.java
│   │   ├── UpdateProductRequest.java
```

```
│   │   ├── LoginRequest.java         // (NOU)
│   │   └── RegisterRequest.java      // (NOU)
│   └── response
│       ├── ProductResponse.java
│       ├── LoginResponse.java        // (NOU)
│       ├── ErrorResponse.java
│       └── ValidationExceptionResponse.java
├── security
│   ├── AppUserDetailsService.java    // (NOU)
│   ├── JwtService.java               // (NOU)
│   ├── JwtAuthenticationFilter.java  // (NOU)
│   ├── RestAuthenticationEntryPoint.java  // (NOU)
│   └── RestAccessDeniedHandler.java  // (NOU)
└── exception
    ├── ProductNotFoundException.java
    └── GlobalExceptionHandler.java
```

11.1. SecurityConfig.java (varianta finală cu JWT)

```
package ro.unitbv.restlab.config;

import lombok.RequiredArgsConstructor;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.authentication.AuthenticationManager;
import
org.springframework.security.authentication.dao.DaoAuthenticationProvider;
import
org.springframework.security.config.annotation.authentication.configuration
.AuthenticationConfiguration;
import
org.springframework.security.config.annotation.method.configuration.EnableM
ethodSecurity;
import
org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticat
ionFilter;
import ro.unitbv.restlab.security.JwtAuthenticationFilter;
import ro.unitbv.restlab.security.RestAccessDeniedHandler;
import ro.unitbv.restlab.security.RestAuthenticationEntryPoint;

@Configuration
@EnableMethodSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtAuthenticationFilter jwtFilter;
    private final RestAuthenticationEntryPoint authEntryPoint;
    private final RestAccessDeniedHandler accessDeniedHandler;
```

```
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public DaoAuthenticationProvider authenticationProvider(
            UserDetailsService userDetailsService,
            PasswordEncoder passwordEncoder) {

        DaoAuthenticationProvider provider =
                                            new DaoAuthenticationProvider();
        provider.setUserDetailsService(userDetailsService);
        provider.setPasswordEncoder(passwordEncoder);
        return provider;
    }

    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
                                                    throws Exception {
        http
            .csrf(csrf -> csrf.disable())
            .cors(cors -> {})
            .sessionManagement(sm -> sm
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/auth/**").permitAll()
                .requestMatchers("/h2-console/**").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/products/**")
                    .hasAnyRole("USER", "ADMIN")
                .requestMatchers("/api/products/**").hasRole("ADMIN")
                .anyRequest().authenticated())
            .exceptionHandling(ex -> ex
                .authenticationEntryPoint(authEntryPoint)
                .accessDeniedHandler(accessDeniedHandler))
            .headers(h -> h.frameOptions(f -> f.disable()))
            .addFilterBefore(jwtFilter,
                             UsernamePasswordAuthenticationFilter.class);

        return http.build();
    }
}
```

11.2. ProductController adnotat cu reguli fine

Autorizarea pe URL declarată în `SecurityConfig` acoperă majoritatea cazurilor. Totuși, pentru claritate și flexibilitate, se pot adăuga și adnotări pe metode [3]:

```
@RestController
@RequestMapping("/api/products")
@RequiredArgsConstructor
@Slf4j
public class ProductController {

    private final ProductService productService;

    @GetMapping
    @PreAuthorize("hasAnyRole('USER', 'ADMIN')")
    public List<ProductResponse> getAllProducts() { /* ... */ }

    @GetMapping("/{id}")
    @PreAuthorize("hasAnyRole('USER', 'ADMIN')")
    public ProductResponse getProductById(@PathVariable Integer id) {
       /* ... */ }

    @PostMapping
    @PreAuthorize("hasRole('ADMIN')")
    public ResponseEntity<ProductResponse> createProduct(
            @Valid @RequestBody CreateProductRequest request) { /* ... */ }

    @PutMapping("/{id}")
    @PreAuthorize("hasRole('ADMIN')")
    public ProductResponse updateProduct(
            @PathVariable Integer id,
            @Valid @RequestBody UpdateProductRequest request) { /* ... */ }

    @DeleteMapping("/{id}")
    @PreAuthorize("hasRole('ADMIN')")
    public ResponseEntity<Void> deleteProduct(@PathVariable Integer id) {
       /* ... */ }
}
```

Observație: Dacă regulile pe URL și cele de metodă se contrazic, verificările se aplică în ordinea filtrelor. Regulile din `SecurityFilterChain` sunt verificate primele, iar dacă blochează cererea, metoda nu mai este apelată și `@PreAuthorize` devine irelevant. Pentru evitarea dublei verificări, se alege o singură abordare consistentă.

12. Testarea endpoint-urilor securizate

Testele de securitate sunt importante pentru a se asigura că regulile de autorizare funcționează corect. Spring Security Test oferă un set de instrumente dedicate. [3]

12.1. Cerere fără autentificare

```
curl -i http://localhost:8080/api/products

HTTP/1.1 401 Unauthorized
Content-Type: application/json

{"timestamp":"...","status":401,"error":"Unauthorized",
 "message":"Full authentication is required...","path":"/api/products"}
```

12.2. Autentificare și obținerea tokenului

```
curl -X POST http://localhost:8080/api/auth/login \
     -H "Content-Type: application/json" \
     -d '{"username":"user","password":"user123"}'

{"token":"eyJhbGciOiJIUzI1NiIs..."}
```

12.3. Utilizator fără rol suficient

```
curl -i -X POST http://localhost:8080/api/products \
     -H "Authorization: Bearer <token-user>" \
     -H "Content-Type: application/json" \
     -d '{"name":"X","price":10,"stock":1}'

HTTP/1.1 403 Forbidden
{"timestamp":"...","status":403,"error":"Forbidden",
 "message":"Access Denied","path":"/api/products"}
```

12.4. Token expirat sau modificat

```
curl -i http://localhost:8080/api/products \
     -H "Authorization: Bearer eyJ... (token manipulat)"

HTTP/1.1 401 Unauthorized
{"status":401,"error":"Unauthorized","message":"..."}
```

13. Probleme frecvente și modul de rezolvare

„All requests authenticated” dar un endpoint rămâne neprotejat

Cauză uzuală: ordinea regulilor. Dacă un `permitAll()` apare înainte de `anyRequest().authenticated()`, funcționează corect. În schimb, plasarea unui matcher după `anyRequest()` este ignorată.

„Access denied” chiar și pentru admin

Cauză uzuală: confuzie `hasRole` / `hasAuthority`. Dacă rolurile sunt stocate fără prefix (`ADMIN`) și apoi se verifică `hasAuthority("ADMIN")`, Spring caută autoritatea `ADMIN`, nu `ROLE_ADMIN`. Soluția este să se folosească consecvent fie `hasRole("ADMIN")` cu permisiune `ROLE_ADMIN`, fie `hasAuthority("PERMISSION_X")` cu permisiuni fără prefix.

Parole stocate care nu pot fi validate

Cauză uzuală: parolă salvată în clar sau cu alt algoritm decât cel așteptat. `BCryptPasswordEncoder.matches(raw, encoded)` verifică dacă `encoded` este un hash BCrypt valid. Dacă nu, întoarce `false` fără excepție, iar autentificarea eșuează cu „Bad credentials”. La inițializare, trebuie codificate parolele **înainte** de salvare.

CORS „funcționează din Postman dar nu din browser”

Postman nu aplică same-origin policy, deci nu trimite preflight. Problema apare strict în browser. Se verifică:

- dacă bean-ul `CorsConfigurationSource` este declarat și `cors()` este apelat în `SecurityFilterChain`;
- dacă originul frontendului coincide exact cu cel din `allowedOrigins` (inclusiv portul);
- dacă nu există un proxy intermediar care elimină antetele CORS.

„JwtFilter” nu este apelat

Dacă filtrul nu este înregistrat cu `addFilterBefore(...)`, sau dacă lanțul `SecurityFilterChain` nu este activat pentru pattern-ul respectiv, filtrul nu se execută. Se verifică, de asemenea, dacă este într-un pachet scanat de Spring.

H2 console afișează pagină goală

Consola H2 este servită într-un iframe; Spring Security adaugă, implicit, antetul `X-Frame-Options: DENY`, care interzice încărcarea. Soluția este dezactivarea protecției pentru consolă:

```
.headers(h -> h.frameOptions(f -> f.disable()))
```

Acest compromis este acceptabil doar în dezvoltare.

14. Întrebări recapitulative

1. Care este diferența, formulată în termeni proprii, între autentificare și autorizare?
2. Ce rol are `DelegatingFilterProxy` și de ce este necesar?
3. În ce ordine se execută filtrele `BasicAuthenticationFilter` și `AuthorizationFilter`? De ce nu pot fi inversate?
4. De ce nu se stochează parolele în clar? De ce `BCrypt` este preferat față de `MD5` sau `SHA-256`?
5. Care este diferența dintre `hasRole("ADMIN")` și `hasAuthority("ADMIN")`?
6. Ce face, exact, `@PreAuthorize` și prin ce mecanism intern este realizată verificarea?
7. De ce protecția CSRF este necesară la formulare web dar poate fi dezactivată la un API stateless cu JWT?
8. Care este diferența dintre `SessionCreationPolicy.STATELESS` și `IF_REQUIRED`?
9. Ce părți are un JWT și care dintre ele este semnată criptografic?
10. Este payload-ul unui JWT criptat? Se pot pune date sensibile în el?
11. Ce face `AuthenticationEntryPoint` și în ce moment este invocat?
12. De ce filtrul JWT custom trebuie plasat înainte de `UsernamePasswordAuthenticationFilter`?

13. Care sunt limitările HTTP Basic și de ce nu este recomandat pentru producție fără HTTPS?
14. Ce este o cerere preflight CORS și când este trimisă?
15. Explicați cum Spring determină dacă un utilizator are permisiunea de a executa o metodă adnotată cu `@PreAuthorize("hasRole('ADMIN')")`.

15. Concluzii

Securitatea unei aplicații web nu este o caracteristică izolată care se adaugă la sfârșit, ci un set de decizii care traversează întreaga arhitectură. În acest laborator s-a pornit de la o aplicație complet deschisă și s-a ajuns la una care:

- impune autentificare pentru orice endpoint protejat;
- stochează parolele sub formă de hash-uri BCrypt, nu în clar;
- autorizează operațiile în funcție de rolurile utilizatorilor, atât pe URL cât și pe metodă;
- funcționează stateless, cu tokenuri JWT, pretându-se la scalare orizontală;
- gestionează corect CORS și CSRF pentru contextul de API;
- returnează erori structurate, în format JSON, pentru cazurile de 401 și 403.

Fiecare dintre aceste decizii reflectă un compromis între securitate, performanță și complexitate. Un developer care înțelege mecanismele din spatele filtrelor, al contextului de securitate și al autentificării poate face aceste compromisuri în cunoștință de cauză, adaptându-le la cerințele concrete ale proiectului.

Spring Security este un framework bogat, cu multe extensii care nu au fost discutate aici (OAuth2 Client, SAML, LDAP, WebAuthn). Cunoașterea solidă a fundamentelor – filtrele, `UserDetailsService`, `PasswordEncoder`, `AuthorizationManager` – permite însă abordarea oricăreia dintre aceste extensii fără dificultate.

16. Exerciții propuse

16.1.1. Exercițiul 1 – Rol nou MANAGER

Se introduce un al treilea rol, `MANAGER`, cu următoarele permisiuni: poate citi toate produsele, poate actualiza prețurile și stocurile (doar aceste câmpuri), dar nu poate crea sau șterge produse. Se vor realiza:

- crearea unui utilizator de test cu rolul `MANAGER`;
- adăugarea regulilor corespunzătoare în `SecurityFilterChain`;
- un endpoint nou `PATCH /api/products/{id}/price` care acceptă doar pe `MANAGER` și `ADMIN`.

Laborator 6: Testarea automată a aplicațiilor REST cu Spring Boot

1. Obiectivul laboratorului

În acest laborator se urmărește înțelegerea **testării automate** a aplicațiilor Spring Boot, cu accent pe **testarea unitară**, dar acoperind sistematic și celelalte tipuri uzuale de teste pe care un programator de aplicații moderne trebuie să le cunoască și să le poată scrie. Punctul de plecare este aplicația REST construită în laboratoarele anterioare (catalogul de produse), pe care se va adăuga, pas cu pas, o suită completă de teste care exersează fiecare strat al arhitecturii: mapper, service, repository, controller, precum și fluxul end-to-end.

La finalul laboratorului, cititorul ar trebui să fie capabil:

- să distingă tipurile fundamentale de testare automată (unitară, de integrare, slice, componentă, end-to-end, regresie, smoke, performanță, contract, mutație) și să spună când și de ce este indicat fiecare;
- să folosească corect terminologia consacrată: SUT, AAA / Given-When-Then, fixture, izolare, code coverage, flaky tests;
- să cunoască taxonomia test doubles a lui Gerard Meszaros (dummy, stub, spy, mock, fake) și să facă diferența esențială dintre mock și stub;
- să utilizeze tehnologiile uzuale ale ecosistemului Spring Boot pentru testare: JUnit 5 (Jupiter), Mockito, AssertJ, MockMvc, slice-urile `@SpringBootTest` / `@WebMvcTest` / `@DataJpaTest`;
- să aleagă în mod conștient între H2 in-memory și Testcontainers pentru testele care ating stratul de persistență;
- să configureze Maven cu Surefire și Failsafe pentru rularea separată a testelor unitare și a celor de integrare, precum și plugin-ul JaCoCo pentru raportul de acoperire cu cod;
- să scrie efectiv teste de complexitate medie pentru toate clasele aplicației REST (mapper, service, controller, exception handler, repository) și să interpreteze corect rezultatele acestora.

2. De ce avem nevoie de testare automată

În stadiul actual, aplicația realizată în laboratoarele anterioare are deja o arhitectură matură: controller-ul primește cererile HTTP, validarea este realizată automat prin Bean Validation, service-ul conține logica de business, repository-ul abstractizează accesul la baza de date, tratarea centralizată a erorilor produce răspunsuri uniforme, iar logging-ul și metricile permit observarea comportamentului în execuție. Totuși, nimic din ceea ce s-a construit până

acum nu garantează că aplicația funcționează corect. Singura modalitate prin care s-a verificat funcționarea a fost **testarea manuală**: pornirea aplicației, trimiterea unor cereri HTTP din browser sau dintr-un client REST și inspecția răspunsurilor. Această abordare este utilă în faza de explorare, dar este complet insuficientă pentru un produs serios.

Testarea manuală are câteva limite fundamentale. În primul rând, este **lentă**: fiecare scenariu trebuie reluat de dezvoltator, ceea ce devine impracticabil pe măsură ce numărul de funcționalități crește. În al doilea rând, este **predispusă la omisiuni**: un caz testat astăzi poate fi uitat săptămâna viitoare, după o modificare aparent neînsemnată. În al treilea rând, este **subiectivă**: două execuții manuale pot să nu acopere exact aceleași scenarii. În al patrulea rând, este **greu de reprodus**: un bug observat ad-hoc, fără pași documentați, poate dispărea fără ca defectul real să fie remediat.

Testarea automată este disciplina care răspunde acestor limite. În esență, este vorba despre scrierea de cod care verifică automat alt cod, urmând un șablon repetabil: se pregătește o stare cunoscută, se execută comportamentul de testat, se compară rezultatul obținut cu cel așteptat. Acest cod poate fi rulat de oricâte ori, în câteva secunde sau minute, fără intervenție umană. El devine, treptat, o plasă de siguranță care permite refactorizarea cu încredere și prevenirea regresiilor.

În practică, testele automate au mai multe roluri. Primul este **verificarea corectitudinii**: confirmă că aplicația face ce trebuie, în condițiile pentru care a fost proiectată. Al doilea este **documentarea executabilă**: un test bine scris descrie clar ce se așteaptă de la o componentă, mai precis decât orice comentariu sau document. Al treilea rol este **protecția față de regresii**: atunci când se adaugă o funcționalitate nouă sau se face o refactorizare, testele existente avertizează imediat dacă vreo bucată de cod care funcționa până atunci a încetat să mai funcționeze. Al patrulea este **orientarea designului**: codul care este greu de testat este aproape întotdeauna și greu de menținut. Atunci când scrierea unui test devine dificilă, motivul este, cel mai adesea, o problemă de design (cuplaj prea strâns, responsabilități amestecate, dependențe ascunse).

Există un argument economic foarte cunoscut, formulat în anii 1980 de Barry Boehm, care arată că costul corectării unui defect crește exponențial cu cât acesta este descoperit mai târziu în ciclul de dezvoltare. Un defect prins prin teste unitare se rezolvă, de regulă, în câteva minute. Același defect descoperit la integrare costă mai mult, iar dacă ajunge la utilizatori, costul devine semnificativ atât în resurse umane, cât și în reputație. Această observație este unul dintre motivele principale pentru care testarea automată este astăzi o practică standard în orice echipă serioasă de dezvoltare software.

3. Tipurile fundamentale de testare automată

Termenul de „testare” este folosit colocvial pentru a denumi orice activitate de verificare a unei aplicații. În realitate, sub această umbrelă există mai multe tipuri distincte de testare, fiecare cu propriul scop, propriul nivel de granularitate, propriile costuri și propriile limitări.

Înțelegerea acestor distincții este esențială: fără ea, o echipă riscă să scrie multe teste care nu acoperă, în fapt, riscurile reale ale aplicației. [25]

Cele mai importante tipuri de testare automată, ordonate aproximativ de la cea mai fină granularitate la cea mai grosieră, sunt:

3.1. Testarea unitară (unit testing)

Testarea unitară verifică o **unitate** izolată de cod. În programarea orientată-obiect, unitatea este, de regulă, o clasă (sau, mai precis, o metodă publică a unei clase) testată independent de colaboratorii ei. Toate dependențele externe – alte clase, baze de date, servicii web, sisteme de fișiere – sunt înlocuite cu **test doubles** (mock-uri, stub-uri, fake-uri etc.), astfel încât testul să se concentreze strict asupra logicii unității respective. [25]

Caracteristicile testelor unitare bune sunt:

- rapide – un test unitar tipic se execută în milisecunde, ceea ce face ca o suită de mii de teste unitare să poată fi rulată în câteva secunde;
- izolate – fiecare test rulează independent de celelalte și nu depinde de o stare globală (bază de date, fișiere, rețea);
- deterministe – același test produce mereu același rezultat, indiferent de momentul rulării;
- focalizate – verifică un singur comportament al unei singure clase;
- auto-validate – testul însuși decide automat dacă a trecut sau a căzut, fără ca dezvoltatorul să interpreteze ieșirea.

Scopul testelor unitare este verificarea logicii interne a unei clase: ramurile if/else, calculele, condițiile la limită, comportamentul în cazuri excepționale. În arhitectura aplicației, candidații naturali pentru testare unitară sunt: `ProductMapper` (transformări pure, fără dependențe), `ProductService` (logica de business, cu dependențe mock-uite), `GlobalExceptionHandler` (mapările dintre excepții și răspunsurile HTTP).

Testele unitare sunt recomandate **oriunde există logică decizională sau transformări de date**. Ele formează baza piramidei testării și ar trebui să fie, ca număr, cele mai multe într-o aplicație matură.

3.2. Testarea de integrare (integration testing)

Testarea de integrare verifică modul în care **mai multe componente colaborează** între ele. Spre deosebire de testele unitare, aici nu se mai izolează strict o singură clasă; se pornește, de regulă, un fragment real din infrastructură (o bază de date, un broker de mesaje, un server web embedded) și se observă comportamentul lanțului complet. [25]

În contextul Spring Boot, un test de integrare clasic pornește contextul aplicației prin adnotarea `@SpringBootTest`, ridică o bază de date H2 in-memory (sau, mai bine, un container Docker prin Testcontainers), expune endpoint-urile pe un port aleator și interacționează cu aplicația prin HTTP real, folosind `TestRestTemplate` sau `WebTestClient`.

Astfel, testul exersează inclusiv: deserializarea JSON-ului de cerere, validarea Bean Validation, traversarea controller → service → repository, traducerea entității în SQL prin Hibernate, executarea SQL pe o bază reală, serializarea răspunsului.

Aceste teste sunt mai lente (uneori câteva secunde fiecare), pentru că pornirea contextului Spring și a infrastructurii nu este gratuită. În compensație, ele oferă o încredere mult mai mare: dacă un test de integrare trece, înseamnă că lanțul de componente, configurația Spring, mapările JPA și schema de bază de date sunt corecte împreună.

Testele de integrare sunt indicate în special pentru: fluxurile critice ale aplicației (CRUD complet pe principalele resurse), comportamente care depind de interacțiunea mai multor straturi (de exemplu, validarea + tratarea centralizată a erorilor), și pentru verificarea contractului HTTP expus de aplicație.

3.3. Slice tests

Slice tests (teste pe secțiuni) reprezintă un compromis specific Spring Boot între unit testing și integration testing. În loc să se pornească întregul context al aplicației, se pornește doar o secțiune verticală a infrastructurii – adică numai componentele necesare pentru testarea unui anumit strat. [1]

Cele mai importante slice tests în Spring Boot sunt:

- `@WebMvcTest` – pornește doar stratul web (Spring MVC, MockMvc, Jackson, Bean Validation, controllere, advice-uri); service-ul, repository-ul, datasource-ul nu sunt încărcate;
- `@DataJpaTest` – pornește doar stratul de persistență (Spring Data, Hibernate, un DataSource embedded); controller-ul și service-ul nu sunt instanțiate;
- `@JsonTest` – pornește doar serializarea/deserializarea Jackson, util pentru testarea DTO-urilor și a configurațiilor de marshaling;
- `@RestClientTest` – pornește doar stratul de client HTTP (RestTemplate, WebClient), util pentru testarea integrării cu API-uri externe.

Slice-urile sunt o invenție elegantă: păstrează multe dintre avantajele testelor unitare (viteză, focalizare, izolare) dar permit testarea componentelor care depind de infrastructura Spring (de exemplu, un controller depinde de mapările Spring MVC, care nu se pot mock-ui ușor). În laboratorul de față vom folosi `@WebMvcTest` pentru `ProductController` și `@DataJpaTest` pentru `ProductRepository`.

3.4. Testarea pe componente (component testing)

Testarea pe componente este un termen folosit mai ales în contextul aplicațiilor distribuite și al microserviciilor. Verifică un singur microserviciu complet – adică toată aplicația ca atare, dar izolată de celelalte microservicii prin mock-uri pentru dependențele externe (alte servicii, brokere de mesaje etc.). Este, în esență, o testare end-to-end limitată la perimetrul unui singur serviciu. Pentru aplicația realizată anterior, testele de integrare joacă, în practică, și rolul de teste componentă.

3.5. Testarea end-to-end (E2E)

Testarea end-to-end verifică sistemul complet, în condiții cât mai apropiate de cele de producție: toate microserviciile rulează, baza de date este reală, interfețele utilizator sunt acționate prin instrumente de automatizare (Selenium, Playwright, Cypress) sau prin clienți HTTP reali. Scopul este validarea **unui scenariu complet de utilizare**, din punctul de vedere al unui actor extern (utilizator, sistem partener). [25]

Avantajele E2E sunt evidente: dacă un astfel de test trece, fluxul respectiv funcționează în întregime. Costul, însă, este foarte ridicat: testele sunt lente (zeci de secunde sau minute), fragile (orice modificare cosmetică în UI le poate strica), greu de depanat (când unul cade, cauza poate fi oriunde în lanț) și predispuse la fenomenul **flaky tests** (teste care, fără modificări de cod, uneori trec, alteori cad – din cauza temporizărilor, a stării reziduale etc.). De aceea, testele E2E trebuie **puține și foarte bine alese**: doar pentru fluxurile esențiale, cu valoare critică pentru afacere.

3.6. Testarea de regresie

Testarea de regresie nu este, în sine, un tip de test cu o tehnologie proprie, ci un **scop** al testelor existente. Spunem că rulăm „o testare de regresie” atunci când re-executăm întreaga suită de teste pentru a verifica dacă modificările recente nu au stricat funcționalități care funcționau înainte. Practic, orice suită bună de teste unitare și de integrare devine, automat, suită de regresie. Acesta este unul dintre cele mai mari beneficii ale testării automate.

3.7. Smoke tests

Smoke tests (sau „teste de fum”, expresie împrumutată din ingineria electronică, unde se pornea un dispozitiv pentru a vedea dacă „scoate fum”) sunt teste foarte largi, dar superficiale, al căror scop este să verifice rapid dacă aplicația pornește și principalele puncte funcționează la nivel minim. În Spring Boot, exemplul canonic este testul `contextLoads()` – generat automat de Spring Initializr – care verifică doar că contextul Spring se încarcă fără erori. Acest test simplu prinde, în practică, surprinzător de multe probleme de configurare.

3.8. Testarea de performanță și de încărcare

Testarea de performanță urmărește timpii de răspuns ai aplicației sub o încărcare dată, iar **testarea de încărcare** (load testing) duce aplicația până la limitele ei pentru a observa cum se comportă. Aceste teste folosesc instrumente specializate (JMeter, Gatling, k6) și se rulează separat de testele funcționale. Ele nu vor fi tratate în acest laborator, dar este important să fie cunoscute ca o categorie distinctă: o aplicație poate fi corectă funcțional, dar inacceptabilă din punct de vedere al performanței.

3.9. Contract testing

Contract testing este un tip de testare folosit în arhitecturile distribuite pentru a verifica faptul că un producător (un microserviciu care expune un API) și un consumator (un alt serviciu care îl apelează) rămân compatibili. Producătorul și consumatorul cad de acord asupra unui contract (o specificație formală a schimbului de date), iar testele de contract verifică, de o parte și de alta, respectarea acestuia. Cele mai cunoscute instrumente sunt Pact și Spring Cloud Contract. Subiectul depășește scopul prezentului laborator, dar este util de știut că, în arhitecturile cu multe servicii, contract testing-ul reduce dependența de teste E2E costisitoare. [25]

3.10. Mutation testing

Mutation testing este o tehnică prin care se verifică calitatea testelor, nu corectitudinea codului. Un instrument specializat (de exemplu, PIT – Pitest pentru Java) generează automat mutanți ai codului testat: variante mici și stricate (înlocuiește `<` cu `<=`, `+` cu `-`, returnează `null` în loc de un obiect valid etc.). Apoi rulează testele asupra fiecărui mutant. Dacă testele nu detectează mutația (mutantul „supraviețuiește”), înseamnă că suita de teste nu este suficient de strictă – un defect similar mutației ar putea trece neobservat. Mutation testing este o investiție utilă pe termen lung: oferă o evaluare mult mai realistă a calității testelor decât simpla acoperire cu cod (code coverage).

3.11. Tabel sintetic al tipurilor de testare

Tabelul de mai jos sintetizează tipurile discutate, comparându-le după criterii relevante pentru alegerea instrumentului potrivit:

Tip	*Granularitate*	*Viteză*	*Stabilitate*	*Când se utilizează*
Unitar	O clasă/metodă	Foarte rapid (ms)	Foarte stabilă	Logică internă, ramuri, cazuri-limită
Slice	Un strat (web, JPA)	Rapid (sute de ms)	Stabilă	Configurare specifică Spring (mapări MVC, JPA)
Integrare	Tot contextul aplicației	Mediu (secunde)	Bună	Fluxuri majore, lanț controller→DB
Componentă	Un microserviciu	Mediu	Bună	Microservicii, izolat de dependențe externe
End-to-end	Sistem complet	Lent (zeci de secunde)	Fragilă	Scenarii de utilizator critice, puține
Smoke	Aplicația pornește	Rapid	Foarte stabilă	Verificare după build/deploy
Performanță	Aplicația sub sarcină	Foarte lent	N/A	Înainte de release, capacity planning
Contract	API între servicii	Rapid	Stabilă	Microservicii, evoluție independentă
Mutație	Calitatea testelor	Foarte lent	N/A	Audit periodic al suitei de teste

4. Piramida testării

O întrebare practică imediată este: câte teste din fiecare tip ar trebui să avem? Nu există un răspuns universal, însă există un model de referință propus de Mike Cohn în cartea *Succeeding with Agile* (2009), cunoscut ca **piramida testării**. Modelul susține că, într-o suită echilibrată, raportul dintre tipurile de teste ar trebui să formeze o piramidă: multe teste unitare la bază, mai puține teste de integrare la mijloc și foarte puține teste end-to-end în vârf.

Justificarea este pragmatică. Testele unitare sunt ieftine de scris, foarte rapide de executat și foarte stabile, deci ne putem permite să avem multe. Testele E2E, în schimb, sunt scumpe de scris, lente de executat și fragile, deci trebuie să fie puține. Testele de integrare se află între cele două extreme.

Pe lângă numărul absolut de teste, piramida sugerează și un principiu important: orice comportament ar trebui testat la cel mai jos nivel posibil. Dacă o regulă de business poate fi verificată printr-un test unitar, este o risipă să o verificăm doar printr-un test E2E. Testul unitar va detecta defectul în câteva milisecunde, în timp ce testul E2E va dura câteva secunde și va localiza defectul mult mai vag.

În practică, există un anti-pattern frecvent întâlnit, denumit ice cream cone (cornetul de înghețată) – piramida întoarsă: foarte multe teste E2E, puține teste de integrare, aproape nimic la nivel unitar, iar deasupra, mai multe teste manuale. Acest tipar produce suite lente, fragile și costisitoare, în care echipa pierde mai mult timp reparând testele decât scriind cod nou. Recunoașterea acestui anti-pattern și migrarea treptată spre o piramidă sănătoasă este una dintre semnele unei discipline de testare mature.

Nivel	***Cantitate (ordin de mărime)***	***Timp tipic / test***	***Observații***
Unit	sute, mii	1-10 ms	Bază solidă, cele mai multe verificări
Integrare / Slice	zeci, sute	0.1-2 s	Verificare a fluxurilor și a configurării
E2E	câteva, zeci	5-60 s	Doar fluxuri critice, atent selectate

5. Concepte și terminologie consacrată

Înainte de a trece la implementare, este necesară fixarea câtorva termeni standard pe care orice programator care scrie teste îi folosește. Aceștia provin din comunitatea internațională (în special din lucrările lui Kent Beck, Martin Fowler și Gerard Meszaros) și sunt aceiași în orice limbaj sau framework.

5.1. System Under Test (SUT)

SUT – abreviere pentru *System Under Test* – desemnează **componenta pe care o testăm** la momentul respectiv. Într-un test unitar pentru `ProductService`, SUT-ul este chiar `ProductService`. Toate celelalte clase implicate (repository-ul, mapper-ul) sunt colaboratori, nu fac parte din SUT și sunt înlocuite cu test doubles. Distincția este utilă: în

orice moment al scrierii unui test, programatorul ar trebui să poată răspunde la întrebarea „cine este SUT-ul aici?”. Confuzia între SUT și colaboratori duce la teste fragile, care cad la modificări nesemnificative.

5.2. Arrange-Act-Assert (AAA) și Given-When-Then

Aproape orice test bun urmează un șablon în trei pași, cunoscut sub două denumiri echivalente: **Arrange-Act-Assert** (AAA) sau **Given-When-Then** (folosit în BDD – Behavior-Driven Development). Cei trei pași sunt:

- Arrange / Given – pregătirea stării: se construiesc obiectele de intrare, se configurează test doubles, se inițializează SUT-ul;
- Act / When – execuția propriu-zisă: se apelează metoda sau comportamentul testat, de regulă o singură dată, asupra SUT-ului;
- Assert / Then – verificarea rezultatului: se compară rezultatul real cu cel așteptat folosind biblioteca de aserțiuni.

Respectarea acestui șablon are mai multe efecte benefice. Un test devine **ușor de citit** – oricine îl deschide identifică rapid ce se pregătește, ce se execută, ce se verifică. Devine, de asemenea, **ușor de menținut** – modificările vizează clar una dintre cele trei zone. Și devine **ușor de extins** – adăugarea unui nou test pentru o variație a aceluiași comportament se face copiind structura existentă și schimbând doar ceea ce diferă. În testele acestui laborator vom marca explicit, prin comentarii, cele trei zone: `// Arrange`, `// Act`, `// Assert`. [25]

5.3. Fixture

Termenul **fixture** se referă la **starea fixă pregătită pentru un test**: setul de obiecte, datele inițiale, configurația. Fixture-ul este construit în pasul Arrange și asigură că testul rulează într-un mediu cunoscut și controlat. Pentru testele complexe, fixture-ul poate fi extras în metode cu adnotarea `@BeforeEach` (rulează înainte de fiecare test) sau `@BeforeAll` (rulează o singură dată, înainte de toate testele clasei). Diferența de semantică este importantă: `@BeforeEach` garantează izolarea, `@BeforeAll` optimizează performanța când fixture-ul este scump de construit, dar partajarea stării între teste poate genera dependențe ascunse.

5.4. Izolarea testelor

Izolarea este principiul conform căruia un test nu trebuie să depindă, în niciun fel, de execuția altui test. Nici de ordinea în care testele sunt rulate, nici de starea lăsată de un test anterior. Încălcarea izolării produce două probleme grave: testele devin imprevizibile (uneori trec, alteori cad, în funcție de ordine), iar depanarea lor devine extrem de dificilă (pentru că reproducerea cere replicarea exactă a contextului). În aplicația noastră, izolarea o vom asigura prin: ștergerea bazei de date înainte de fiecare test (`@BeforeEach` cu `productRepository.deleteAll()`), tranzacții cu rollback automat (`@DataJpaTest` face acest lucru implicit), folosirea de `@DirtiesContext` când contextul Spring trebuie reîmprospătat. [25]

5.5. Code coverage

Code coverage (acoperire cu teste) este un indicator care măsoară procentul de cod sursă care este executat în timpul rulării testelor. Există mai multe variante de coverage:

- line coverage – câte linii din codul sursă au fost atinse de cel puțin un test;
- branch coverage – câte ramuri ale instrucțiunilor decizionale (if/else, switch, ternar) au fost executate;
- method coverage – câte metode au fost apelate cel puțin o dată;
- class coverage – câte clase au cel puțin o linie executată.

În ecosistemul Java, instrumentul standard este **JaCoCo** (Java Code Coverage). Acesta instrumentează bytecode-ul în timpul rulării și produce un raport detaliat, navigabil clasă cu clasă, metodă cu metodă, linie cu linie.

Există însă o atenționare importantă: code coverage nu este o măsură a calității testelor. Un test poate atinge o linie fără a verifica nimic util. Acoperirea de 100% este perfect compatibil cu o suită de teste fără aserțiuni serioase. În practică, se urmărește o acoperire rezonabil (60-80% pentru proiecte uzuale, mai mult pentru proiecte critice), iar importantă este **calitatea aserțiunilor**, nu doar valoarea procentuală. [25]

5.6. Flaky tests

Flaky tests sunt teste care, fără modificări de cod, uneori trec, alteori cad. Cauzele tipice sunt: dependența de ordinea de execuție, dependența de timp (test care folosește `Thread.sleep` în loc de o așteptare deterministă), dependența de un sistem extern (rețea, ceas de sistem, generator pseudoaleator), partajarea de stare între teste (un test poluează baza de date, altul se așteaptă la o stare goală).

Flaky tests sunt extrem de costisitoare: erodează încrederea echipei în suita de teste. Atunci când un test cade, prima reacție devine „probabil e flaky, repornesc build-ul”, în loc de „verific imediat ce am stricat”. În acel moment, suita de teste încetează să mai protejeze produsul. Tratarea fermă a flaky tests (depanare imediată sau eliminare) este o regulă esențială de igienă într-un proiect serios.

5.7. Test smell

Test smell este un termen analog *code smell*: un semnal că un test nu este bine scris, chiar dacă funcționează. Câteva mirosuri tipice: aserțiuni lipsă sau insuficiente, denumiri vagi (`test1`, `testService`), un singur test care verifică zece comportamente, mock-uri care întorc alte mock-uri, stub-uri configurate dar niciodată folosite, fixture-uri imense și opace. Un test cu smell rămâne în suită, dar contribuie tot mai puțin pe măsură ce trece timpul.

6. Test Doubles – taxonomia lui Meszaros

Atunci când se scrie un test unitar pentru o clasă cu dependențe (`ProductService` depinde de `ProductRepository` și de `ProductMapper`), trebuie să înlocuim aceste dependențe cu

obiecte controlate de test. Termenul generic pentru aceste înlocuitori este **test double** (analog cu „cascadorul” din film, care îl dublează pe actor). Termenul a fost introdus de Gerard Meszaros în cartea *xUnit Test Patterns* (2007), care a propus o taxonomie cu cinci categorii. Această taxonomie este astăzi standardul de referință în literatura de testare și este folosită identic în Java, C#, Python, Ruby etc. [25]

6.1. Dummy

Un **dummy** este un obiect care este transmis ca parametru, dar nu este folosit niciodată. Există doar pentru a satisface o semnătură de metodă. Tipic, este folosit pentru a completa parametri obligatorii într-un constructor sau într-o metodă, fără ca testul să se intereseze de comportamentul lor. Un `null` ar putea fi suficient adesea, dar dacă obiectul este folosit chiar și pentru o verificare de `null`, este nevoie de un dummy real (un obiect gol).

6.2. Stub

Un **stub** este un obiect care furnizează răspunsuri fixe pentru apelurile pe care le primește. Este folosit pentru a izola SUT-ul de comportamentul real al colaboratorilor și pentru a stabili o stare de intrare cunoscută în test. Nu se verifică interacțiunile cu un stub – se verifică doar efectul asupra SUT-ului.

Exemplu: într-un test pentru `ProductService.findById(42)`, configurăm repository-ul ca stub: `when(productRepository.findById(42)).thenReturn(Optional.of(produs))`. Repository-ul nu mai face nimic cu o bază reală; răspunde direct cu valoarea pe care i-o impunem. Apoi verificăm că service-ul a returnat corespondentul mapat al acelui produs. Aici repository-ul joacă rolul de stub: ne interesează ce returnează, nu cum a fost apelat.

6.3. Spy

Un **spy** este un obiect care **înregistrează apelurile primite** pentru a putea fi inspectate ulterior. Spre deosebire de stub, un spy păstrează un istoric al interacțiunilor (cu ce parametri a fost apelat, de câte ori, în ce ordine). În unele biblioteci, un spy este un mic decorator al unui obiect real: păstrează comportamentul real, dar adaugă posibilitatea de a verifica apelurile. În ecosistemul Java, Mockito face distincția dintre `@Mock` (creează un obiect complet artificial) și `@Spy` (împachetează un obiect real).

6.4. Mock

Un **mock** este un obiect preprogramat cu așteptări: testul declară, înainte de execuție, ce apeluri trebuie să primească mock-ul (și cu ce parametri). În timpul rulării, mock-ul înregistrează apelurile reale, iar la final testul verifică automat dacă așteptările au fost îndeplinite. Un mock combină capabilitățile de stub (poate returna valori fixe) cu cele de spy (înregistrează apelurile). [14]

Important: în terminologia Mockito, orice obiect creat cu `@Mock` sau `Mockito.mock()` este, tehnic, un mock. Dar dacă pe el doar se configură `when().thenReturn()` fără a se mai face `verify()`, se folosește, conceptual, ca pe un stub. Această ambiguitate de terminologie

este sursă constantă de confuzie, motiv pentru care unii autori folosesc termenul generic **test double** atunci când distincția nu este esențială.

6.5. Fake

Un **fake** este o implementare alternativă reală, dar simplificată a unei dependențe. Nu este nici stub, nici mock – este chiar o implementare funcțională, dar potrivită doar pentru utilizare în teste, nu în producție. Exemple clasice de fake-uri:

- bază de date H2 in-memory care înlocuiește o bază MySQL/Postgres reală în teste;
- un repository care păstrează produsele într-un HashMap, care implementează interfața `ProductRepository`, dar fără să atingă vreo bază de date;
- implementare de `SimpleMeterRegistry` pentru Micrometer, care păstrează metricile în memorie pentru a fi inspectate într-un test.

Avantajul unui fake este că oferă comportament real, lucru util când testarea cu mock-uri ar fi prea fragilă (mock-urile reproduc, în esență, contractul colaboratorului – dacă acel contract are subtilități, mock-ul riscă să le ignore). Dezavantajul este complexitatea: a construi un fake corect necesită efort. De aceea, fake-urile sunt scrise și menținute, în general, ca biblioteci comune (H2, `SimpleMeterRegistry`, `MockWebServer`).

6.6. Sinteza taxonomiei

Tip	*Caracteristică principală*	*Exemplu în acest laborator*
Dummy	Obiect inert, doar pentru a umple un parametru.	(nu este folosit explicit)
Stub	Returnează valori fixe pentru apelurile primite.	`ProductRepository` configurat cu `when().thenReturn()` în testele `ProductService`.
Spy	Înregistrează apelurile reale pe un obiect real.	(nu este folosit; Mockito `@Mock` acoperă rolul)
Mock	Stub + verificarea apelurilor așteptate.	Mock-urile Mockito pe care apoi se apelează `verify()`, `times()`, `never()`.
Fake	Implementare alternativă reală, simplificată.	`SimpleMeterRegistry` pentru metrici; H2 in-memory pentru baza de date.

7. Mock vs Stub – distincția esențială

Distincția dintre **mock** și **stub** este atât de importantă încât merită discutată separat. Sursa cea mai citată este articolul lui Martin Fowler, *Mocks Aren't Stubs* (2007), care a fixat în comunitatea profesională această diferență. [14, 25]

Diferența nu este la nivelul **obiectului**, ci la nivelul **stilului de testare**:

- Stub-ul țintește verificarea STĂRII: după ce rulează SUT-ul, ne uităm la rezultat (valoarea returnată, starea finală a unui obiect) și o comparăm cu cea așteptată. Stub-ul ne ajută doar să stabilim o stare de intrare cunoscută.

- Mock-ul țintește verificarea COMPORTAMENTULUI: după ce rulează SUT-ul, ne uităm la INTERACȚIUNILE pe care le-a avut cu colaboratorii (a apelat metoda X cu parametrul Y? de câte ori? în ce ordine?). Aici nu ne mai interesează atât valoarea returnată, cât drumul pe care l-a urmat SUT-ul.

În practică, cele două stiluri se combină frecvent. Pentru `ProductService.create()`, de exemplu, vom verifica:

- starea: response-ul returnat conține datele așteptate (state-based);
- comportamentul: repository-ul a fost apelat cu metoda save() și entitatea corectă (interaction-based).

O regulă utilă este: **preferă verificarea de stare oricând este posibilă**. Verificarea comportamentală cuplează testul de implementarea internă (de exemplu, „știe" că service-ul folosește metoda `save` a repository-ului). Dacă mâine vom înlocui acea metodă cu `saveAndFlush`, testul cade, deși comportamentul observabil al aplicației nu s-a schimbat. Verificarea de stare nu are această fragilitate. Verificarea comportamentală este indispensabilă acolo unde nu există stare observabilă (de exemplu, atunci când unitatea trimite o notificare prin email – singurul lucru pe care îl putem verifica este faptul că a fost apelată metoda `send`).

7.1. Anti-patternul „test ca oglindă a implementării"

O capcană frecventă în testarea cu mock-uri este suprasaturarea de `verify()`: testul ajunge să descrie pas cu pas exact ce face implementarea. Acest tip de test este cuplat strâns cu codul SUT-ului: orice refactorizare, fie ea și pur internă, strică testul. Practic, testul nu mai protejează comportamentul (ce face SUT-ul observabil), ci implementarea (cum face). Pe termen lung, asemenea teste devin o povară: programatorii ezită să refactorizeze codul, pentru că orice schimbare cere și actualizarea testelor. [14]

Regula de echilibru este: verificați comportamentul observabil de la exterior (intrări → ieșiri, interacțiuni cu lumea exterioară), nu detaliile interne. Dacă o linie de cod poate fi modificată fără ca utilizatorul să simtă diferența, ea ar trebui să poată fi modificată fără ca testele să cadă.

8. Tehnologii pentru testare în ecosistemul Spring Boot

Spring Boot vine, începând cu versiunea 1.4, cu un starter dedicat pentru testare: `spring-boot-starter-test`. Acesta agregă, într-o singură dependență, un set complet de biblioteci verificate de comunitate, cu versiuni compatibile între ele. Pentru proiectele Spring Boot, este recomandat să nu se adauge manual versiuni ale acestor biblioteci, ci să se folosească tot ceea ce vine prin starter. [1, 13]

Bibliotecile incluse în spring-boot-starter-test sunt:

- JUnit 5 – framework-ul fundamental pentru organizarea și rularea testelor;
- Mockito – biblioteca de creare și control al test doubles (mock-uri, spy-uri);

- AssertJ – biblioteca de aserțiuni fluente, cu mesaje de eroare expresive;
- Hamcrest – biblioteca de matchere (folosită mai ales prin MockMvc);
- JSONPath și JSONassert – pentru aserțiuni pe documente JSON;
- Spring Test și Spring Boot Test – infrastructura Spring pentru integrarea testelor cu contextul aplicației: `@SpringBootTest`, `@WebMvcTest`, `@DataJpaTest`, `MockMvc`, `TestRestTemplate`.

8.1. JUnit 5 (Jupiter)

JUnit 5 (numit și **Jupiter**, în conformitate cu numele intern al modulelor) este, începând cu 2017, framework-ul standard de testare în Java. Are o arhitectură modulară împărțită în trei mari componente:

- JUnit Platform – fundația; oferă API-ul de descoperire și lansare a testelor; este folosit de IDE-uri (IntelliJ, Eclipse) și de unelte de build (Maven Surefire);
- JUnit Jupiter – modulul cu API-urile noi de scriere a testelor (`@Test`, `@BeforeEach`, `@Nested`, `@ParameterizedTest`, etc.);
- JUnit Vintage – un motor de compatibilitate care permite rularea testelor JUnit 4 vechi pe noua platformă, util în migrarea proiectelor existente.

Adnotările uzuale ale JUnit 5 sunt:

Adnotare	***Rol***	***Echivalent JUnit 4***
`@Test`	Marchează metoda ca test.	`@Test`
`@BeforeEach`	Rulează înainte de fiecare test din clasă.	`@Before`
`@AfterEach`	Rulează după fiecare test din clasă.	`@After`
`@BeforeAll`	Rulează o singură dată, înainte de toate testele.	`@BeforeClass`
`@AfterAll`	Rulează o singură dată, după toate testele.	`@AfterClass`
`@DisplayName`	Atribuie un nume descriptiv testului.	(lipsă)
`@Nested`	Permite gruparea testelor în clase interne.	(lipsă)
`@Disabled`	Dezactivează temporar un test.	`@Ignore`
`@ParameterizedTest`	Test rulat repetat cu seturi diferite de date.	(lipsă, exista `@Theory` limitat)
`@ExtendWith`	Atașează o extensie (de ex. MockitoExtension).	`@RunWith`

Adnotarea `@DisplayName` merită o mențiune specială. Permite atribuirea unui nume descriptiv, în limbaj natural, oricărei metode de test sau clase de teste. Numele apare în rapoartele Maven și în IDE-uri, în locul numelui metodei. Acesta transformă suita de teste într-o documentație executabilă: un raport de execuție citit de cineva care nu a scris codul devine inteligibil. În laboratorul de față, aproape fiecare test va avea un `@DisplayName` explicit, în limba română. [13]

8.2. AssertJ

AssertJ este o bibliotecă de aserțiuni fluente: expresiile de verificare se citesc aproape ca o frază. Punctul de plecare este metoda statică `assertThat()`, care produce un obiect-assertor specializat pe tipul argumentului (string, listă, număr, hartă, optional etc.), cu metode adaptate.

Câteva exemple comparative cu JUnit clasic:

```
// JUnit 5 standard
assertEquals("Tastatura", product.getName());
assertTrue(list.size() == 3);
assertTrue(list.contains("X"));

// AssertJ - mai expresiv si cu mesaje de eroare mai bune
assertThat(product.getName()).isEqualTo("Tastatura");
assertThat(list).hasSize(3).contains("X");
assertThat(map).containsEntry("name",
"Tastatura").doesNotContainKey("price");
assertThat(optional).isPresent().hasValueSatisfying(p ->
assertThat(p.getId()).isEqualTo(1));
```

În plus, AssertJ produce mesaje de eroare semnificativ mai utile. Atunci când o aserțiune cade, mesajul indică exact ce diferă: ce conținea valoarea reală, ce se aștepta, ce element lipsește dintr-o colecție etc. Pentru a înțelege rapid de ce un test a căzut, AssertJ economisește mult timp comparativ cu JUnit clasic. Din aceste motive, toate testele acestui laborator folosesc AssertJ pentru aserțiunile principale.

8.3. Mockito

Mockito este biblioteca de facto pentru crearea și controlul test doubles în Java. Permite crearea de obiecte mock dintr-o clasă sau interfață, configurarea răspunsurilor lor (`when().thenReturn()`) și verificarea apelurilor primite (`verify()`). Sintaxa este declarativă și inteligibilă. Mockito este integrat nativ cu JUnit 5 prin extensia `MockitoExtension`, care se atașează cu adnotarea `@ExtendWith(MockitoExtension.class)`. [14]

Adnotările principale ale Mockito sunt:

- `@Mock` – creează un mock al unei interfețe sau clase; toate metodele returnează valori implicite (null, 0, false, listă goală) până când se configurează altfel;
- `@Spy` – creează un spy în jurul unui obiect real; metodele apelează implementarea reală, dar pot fi suprascrise individual;
- `@InjectMocks` – instanțiază clasa testată și încearcă să injecteze mock-urile @Mock în constructori, setteri sau câmpuri;
- `@Captor` – creează un `ArgumentCaptor` pentru a captura argumentele primite de o metodă mockată.

Metodele esențiale ale API-ului static sunt:

```
// Configurarea unui stub:
when(repository.findById(1)).thenReturn(Optional.of(product));
when(repository.findById(99)).thenReturn(Optional.empty());

// Configurarea unui stub care arunca o exceptie:
when(repository.findById(99)).thenThrow(new RuntimeException("..."));

// Pentru metode void:
doNothing().when(service).deleteById(1);
doThrow(new ProductNotFoundException(99)).when(service).deleteById(99);

// Verificarea apelurilor:
verify(repository).save(product);              // exact o data
verify(repository, times(2)).findAll();        // de exact doua ori
verify(repository, never()).deleteAll();       // niciodata
verify(repository, atLeastOnce()).existsById(anyInt());

// Capturarea argumentelor:
ArgumentCaptor<Product> captor = ArgumentCaptor.forClass(Product.class);
verify(repository).save(captor.capture());
Product captured = captor.getValue();
assertThat(captured.getName()).isEqualTo("Test");
```

Notă privind versiunile: în Spring Boot 3.4+ a fost introdusă adnotarea `@MockitoBean`, care înlocuiește treptat `@MockBean` (deprecated în Spring Boot 3.4). Proiectul nostru folosește Spring Boot 3.3.5, deci vom utiliza `@MockBean`; în proiectele care vor migra la 3.4+, recomandarea este înlocuirea cu `@MockitoBean` și `@MockitoSpyBean`.

8.3.1. Argument matchers: any(), eq(), anyInt() și asociații

În loc de valori efective se pot utiliza expresii precum `any()`, `anyInt()`, `any(Product.class)`. Acestea sunt **argument matchers** – primitive Mockito care înlocuiesc o valoare concretă atunci când configurăm un stub sau verificăm un apel. Deși par un detaliu de sintaxă, înțelegerea corectă a lor este esențială pentru a evita o categorie întreagă de erori care apar frecvent la cei care încep să folosească Mockito.

De ce sunt necesare

Atunci când stubăm o metodă cu `when(repo.findById(1)).thenReturn(...)`, Mockito înregistrează stub-ul pentru **exact** argumentul `1`: orice alt apel cu un alt argument va returna valoarea implicită (`Optional.empty()` în acest caz). Acest comportament este corect și predictibil. Problema apare când în SUT nu putem prezice argumentul exact: o entitate `Product` construită intern de mapper, un timestamp generat în service, un UUID aleator. Pentru aceste cazuri avem nevoie de **matchere** – predicate care indică „acceptă orice argument de un anumit tip” sau „acceptă orice valoare care satisface o condiție”.

Matcherele uzuale

Matcher	*Semnificație*	*Exemplu*
`any()`	Orice valoare, inclusiv `null`. Tipul nu este verificat.	`verify(repo).save(any())`
`any(Class.class)`	Orice instanță (nenula) a clasei date.	`verify(repo).save(any(Product.class))`
`anyInt()`, `anyLong()`, `anyDouble()`, `anyBoolean()`	Orice valoare primitivă (nu acceptă `null`).	`when(repo.findById(anyInt())).thenReturn(...)`
`anyString()`	Orice șir nenul.	`verify(logger).warn(anyString())`
`anyList()`, `anyMap()`, `anySet()`	Orice colecție nenulă.	`verify(repo).saveAll(anyList())`
`eq(value)`	Egalitate cu valoarea dată (folosind `.equals()`).	`verify(service).update(eq(1), any())`
`argThat(predicate)`	Argument care satisface un predicat custom.	`verify(repo).save(argThat(p -> p.getPrice() > 100))`

> **Regula „totul sau nimic”**
>
> Există o regulă strictă pe care Mockito o impune și pe care mulți programatori începători o încalcă: **dacă se folosește un matcher pentru unul dintre argumente, trebuie să se folosească matchere pentru TOATE argumentele apelului**. Nu se pot amesteca valori concrete cu matchere.

Iată exemple de cod corect și incorect:

```
// GRESIT - mix de matcher si valoare concreta:
verify(productService).update(1, any(UpdateProductRequest.class));
//                     ^ valoare concreta + matcher la pozitia 2
// Mockito arunca: InvalidUseOfMatchersException

// CORECT - eq() impacheteaza valoarea concreta intr-un matcher:
verify(productService).update(eq(1),
                              any(UpdateProductRequest.class));

// CORECT - daca toate argumentele sunt concrete, fara matchere:
verify(productService).update(1, specificRequest);

// CORECT - daca toate sunt matchere:
verify(productService).update(anyInt(),
any(UpdateProductRequest.class));
```

Acesta este motivul pentru care `eq()` există: este matcherul care reprezintă „exact această valoare” și permite combinarea valorilor concrete cu alte matchere în același apel. Erorile `InvalidUseOfMatchersException` – care apar adesea în mesaje confuze, lăsând impresia unei probleme de configurare – au, de regulă, această cauză.

eq() vs valoare concretă

Atunci când nu există nicio constrângere de combinare, `eq(1)` și `1` produc același efect – ambele cer ca argumentul să fie egal cu `1`. Convenția uzuală este: pentru claritate, folosim valori concrete acolo unde toate argumentele sunt concrete (`verify(repo).save(product)`), și `eq()` doar acolo unde nevoia de a-l combina cu un alt matcher îl face obligatoriu.

argThat() pentru condiții complexe

Atunci când nu este suficient să verificăm tipul argumentului, ci o proprietate a lui (de exemplu, „produsul salvat are prețul mai mare decât 100”), folosim `argThat()` cu un predicat lambda:

```
// Verifica salvarea unui produs cu pret > 100:
verify(productRepository).save(argThat(product ->
    product.getPrice() > 100 && product.getName().startsWith("Premium")
));

// In stub-uri, similar:
when(productRepository.save(argThat(p -> p.getId() == null)))
    .thenReturn(savedProduct);
```

Comparat cu `ArgumentCaptor`: `argThat()` este mai potrivit când vrem să exprimăm o condiție pe care argumentul trebuie să o satisfacă (test eșuează pe loc dacă nu o satisface), iar `ArgumentCaptor` este mai potrivit când vrem să capturăm argumentul pentru aserțiuni AssertJ ulterioare, mai elaborate (cu mesaje de eroare mai informative). În testele acestui laborator au fost folosite ambele tehnici, exact în acest spirit.

Greșeala clasică: matcher folosit în afara when/verify

Matcherele Mockito **nu sunt expresii care produc valori**. Sub capotă, fiecare apel `any()` sau `eq(...)` înregistrează o intrare într-o stivă internă a Mockito și returnează o valoare implicită (`null` sau `0`), folosită doar pentru a satisface compilatorul. Folosirea unui matcher **în afara** unui apel `when()` sau `verify()` este o eroare frecventă, care poate produce comportamente foarte greu de depanat:

```
// GRESIT - any() folosit ca o "valoare":
Product result = any(Product.class);   // returneaza null si polueaza stiva
service.process(result);

// CORECT - any() apare doar in contextul Mockito:
when(repo.findById(anyInt())).thenReturn(Optional.empty());
verify(repo).save(any(Product.class));
```

Mesajele de eroare produse de această greșeală sunt confuze, pentru că Mockito raportează problema mult mai târziu, la următorul `verify()` sau la curățarea contextului – moment în care contextul real al greșelii s-a pierdut.

8.4. Spring Boot Test slices și @SpringBootTest

Pentru testele care implică infrastructura Spring (controllere, repository-uri, configurări), Spring Boot oferă două categorii de adnotări: `@SpringBootTest` (pornește contextul complet) și test slices (pornesc doar o parte). [1]

8.4.1. @SpringBootTest

`@SpringBootTest` pornește întreg contextul Spring al aplicației, ca și cum aplicația ar fi rulat normal. Are mai multe variante de configurare a serverului web:

- `WebEnvironment.MOCK` (implicit) – nu pornește un server real; controllerele sunt expuse printr-un `MockMvc`;
- `WebEnvironment.RANDOM_PORT` – pornește un server Tomcat embedded pe un port aleator; testul poate folosi `TestRestTemplate` sau `WebTestClient` pentru interacțiuni HTTP reale;
- `WebEnvironment.DEFINED_PORT` – pornește serverul pe portul configurat în application.properties;
- `WebEnvironment.NONE` – fără server web (util pentru aplicații care nu sunt servere web).

Pentru testele de integrare end-to-end, se va folosi `WebEnvironment.RANDOM_PORT`: portul aleator evită conflictele când mai multe teste rulează în paralel sau când portul implicit (8080) este ocupat de altceva.

8.4.2. @WebMvcTest

`@WebMvcTest` pornește doar stratul web. Sunt incluse: serializatorii Jackson, infrastructura de validare (Bean Validation), `MockMvc`, controlerele și `@ControllerAdvice` / `@RestControllerAdvice`. Service-urile, repository-urile, configurările generale nu sunt încărcate. Acest lucru face ca testele să fie foarte rapide (sute de ms), dar înseamnă că dependențele service-ului trebuie furnizate cu `@MockBean`. Adnotarea acceptă un argument: `@WebMvcTest(ProductController.class)` încarcă doar acest controller, ignorându-i pe ceilalți. Astfel, fiecare test focalizează pe un singur controller.

8.4.3. @DataJpaTest

`@DataJpaTest` pornește doar stratul de persistență: Spring Data, Hibernate, un DataSource embedded (H2 implicit, dacă există pe classpath). Infrastructura web nu este încărcată. Fiecare test rulează într-o tranzacție cu rollback automat la final, ceea ce asigură izolarea perfectă între teste fără să fie nevoie de curățare manuală a bazei. Beneficiul colateral este și viteza: bazele de date in-memory pornesc în zecimi de secundă.

În interiorul unui `@DataJpaTest`, este disponibil un bean util numit `TestEntityManager` – un wrapper zvelt peste `EntityManager` care expune metode dedicate testelor (`persistAndFlush`, `find`, `clear`). Avantajul este că putem persista o entitate ocolind repository-ul (deci fără a fi influențați de logica acestuia), ceea ce este foarte util pentru a pregăti starea în pasul Arrange.

8.5. MockMvc

`MockMvc` este un instrument oferit de Spring Test care simulează stratul Spring MVC fără a porni un server real. Cererile HTTP sunt construite prin API fluent (`get("/api/products")`, `post(...)`, `put(...)` etc.), trimise printr-un dispatch intern și răspunsurile sunt verificate prin aserțiuni (`status()`, `jsonPath()`, `content()`). Avantajul este viteza foarte mare: nu există rețea, nu există serializare TCP, doar trecerea prin stiva MVC. Testele `MockMvc` rulează tipic în câteva milisecunde fiecare. [2]

Un exemplu de utilizare:

```
mockMvc.perform(get("/api/products/1"))
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.id").value(1))
    .andExpect(jsonPath("$.name").value("Tastatura"));

mockMvc.perform(post("/api/products")
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(request)))
    .andExpect(status().isCreated())
    .andExpect(jsonPath("$.id").exists());
```

Expresiile `$.id` sunt expresii JSONPath: o sintaxă similară cu XPath, dar pentru documente JSON. `$` reprezintă rădăcina documentului, `.id` accesează un câmp, `[0]` indexează în array, `..*` parcurge recursiv. JSONPath este foarte expresiv și permite aserțiuni țintite asupra structurii JSON, fără a deserializa întregul răspuns într-un obiect Java.

8.6. TestRestTemplate

`TestRestTemplate` este, în schimb, un client HTTP real adaptat pentru testare. Spre deosebire de `MockMvc`, el produce conexiuni TCP reale către un server pornit de `@SpringBootTest(webEnvironment = RANDOM_PORT)`. Rezultatul este un test mai apropiat de scenariul real (cererile trec prin Tomcat, prin serializatori, prin întregul lanț), dar și mai lent. [1]

Se vor folosi `TestRestTemplate` pentru testele de integrare end-to-end. Pentru testele specializate pe stratul web (slice tests), `MockMvc` este alegerea potrivită.

8.7. H2 vs Testcontainers

Pentru testele care ating stratul de persistență, există două abordări principale: o bază H2 in-memory sau o bază reală (MySQL, PostgreSQL) pornită într-un container Docker prin Testcontainers.

Comparația sintetică este:

Criteriu	***H2 in-memory***	***Testcontainers***
Viteză de pornire	Sub 1 secundă	5-15 secunde la prima rulare (descărcare imagine + start)
Fidelitate față de producție	Aproximativă (dialect emulat)	Identică (același engine ca în producție)

Criteriu	*H2 in-memory*	*Testcontainers*
Dependențe externe	Niciuna	Necesită Docker pe mașina de build
Risc de incompatibilități SQL	Crescut	Inexistent
Recomandat pentru	Slice tests, multe teste rapide	Teste de integrare critice, înainte de release

Diferența cea mai importantă este fidelitatea. H2 emulează MySQL sau PostgreSQL prin moduri de compatibilitate, dar emularea nu este completă. Există funcții SQL specifice (de exemplu, `DATE_FORMAT` în MySQL, `array_agg` în PostgreSQL), tipuri proprii (JSON, INET, intervale temporale), comportamente la limită (ordonarea valorilor NULL, sensibilitatea la majuscule), pe care H2 le tratează diferit. Un cod care trece testele cu H2 poate eșua în producție pe MySQL real. [1]

Strategia recomandată în proiectele moderne este ambele: H2 pentru `@DataJpaTest` și pentru testele rapide care nu folosesc particularități ale bazei, Testcontainers pentru testele de integrare critice, rulate în faza `verify` a build-ului. Această împărțire echilibrează viteza pentru ciclul de dezvoltare cu încrederea pentru release-uri. Se vor configura ambele în acest laborator.

8.8. Testcontainers

Testcontainers este o bibliotecă Java care permite pornirea oricărui container Docker ca dependență a unui test JUnit. Containerul pornește înainte de teste, este accesibil prin metode (`getJdbcUrl()`, `getMappedPort()` etc.) și se oprește automat la final. Există module pre-configurate pentru baze de date relaționale (MySQL, PostgreSQL, Oracle, SQL Server), baze NoSQL (MongoDB, Cassandra), brokere (Kafka, RabbitMQ, ActiveMQ), cache-uri (Redis), engine-uri de căutare (Elasticsearch). Pentru fiecare modul, există o clasă Java care expune un container deja configurat (de exemplu, `MySQLContainer`), iar aceasta ascunde complexitatea sintaxei Docker. [1]

Începând cu Spring Boot 3.1, integrarea cu Testcontainers a devenit cu mult mai simplă prin adnotarea `@ServiceConnection` (din pachetul `org.springframework.boot.testcontainers.service.connection`). Aceasta permite legarea automată a contextului Spring la container-ul declarat, fără a mai scrie manual proprietăți `@DynamicPropertySource`. Codul devine extrem de concis.

Un test cu MySQL real prin Testcontainers arată astfel:

```
@SpringBootTest(webEnvironment = WebEnvironment.RANDOM_PORT)
@Testcontainers
class ProductTestcontainersIT {

    @Container
    @ServiceConnection
    static MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0");

    @Autowired
    private TestRestTemplate restTemplate;
```

```
    @Test
    void shouldCreateAndRetrieveProduct() {
        // Folosim aplicatia ca pe orice test de integrare;
        // Spring Boot a configurat automat datasource-ul catre MySQL real.
        // ...
    }
}
```

Trei aspecte merită subliniate:

- `@Container` marchează un container care va fi pornit automat înainte de teste de către extensia Testcontainers a JUnit 5;
- `@ServiceConnection` (Spring Boot 3.1+) face ca Spring Boot să detecteze automat container-ul și să configureze DataSource-ul pe baza lui – fără a mai fi nevoie să scriem proprietăți explicite;
- modificatorul static al variabilei mysql este important: indică un singur container partajat între toate testele clasei (eficiență); fără static, container-ul ar fi recreat la fiecare metodă de test.

Cerința hardware pentru rularea testelor Testcontainers este existența unei instalări de Docker pe mașina pe care rulează build-ul. În medii de CI/CD (GitLab CI, GitHub Actions, Jenkins), Docker este, de regulă, disponibil; în setup-ul local de dezvoltare, programatorii trebuie să aibă instalat Docker Desktop sau echivalent.

9. Pregătirea proiectului pentru testare

Înainte de a scrie efectiv testele, trebuie să configurăm proiectul Maven în mod corespunzător. Acoperim aici trei aspecte: structura directoarelor, dependențele pom.xml și plugin-urile de build pentru rularea testelor.

9.1. Structura directoarelor

Convenția standard Maven separă codul de producție de codul de teste prin două ierarhii paralele: `src/main/java` și `src/test/java`. Aceeași separare este aplicată și pentru resursele non-Java (`src/main/resources` vs. `src/test/resources`). Această convenție este respectată automat de Maven, IntelliJ IDEA și Eclipse, fără configurare suplimentară. Esențial: clasele de test au acces la clasele din `src/main`, dar inversa nu este permisă – nicio clasă de producție nu poate face import dintr-o clasă de test, ceea ce previne accidentele. [1, 13]

Două convenții importante de denumire a fișierelor de test sunt:

- clase care se termină în Test, *Tests sau Test* – sunt rulate de plugin-ul maven-surefire-plugin în faza test (acestea sunt testele unitare și slice tests);
- clase care se termină în IT (Integration Test) – sunt rulate de plugin-ul maven-failsafe-plugin în faza verify (acestea sunt testele de integrare costisitoare, cum ar fi cele cu Testcontainers).

Structura directoarelor proiectului nostru, după adăugarea testelor, va fi:

```
restlab/
├── pom.xml
└── src/
    ├── main/
    │   ├── java/
    │   │   └── ro/unitbv/restlab/
    │   │       ├── RestLabApplication.java
    │   │       ├── controller/
    │   │       │   └── ProductController.java
    │   │       ├── service/
    │   │       │   └── ProductService.java
    │   │       ├── repository/
    │   │       │   └── ProductRepository.java
    │   │       ├── model/
    │   │       │   └── Product.java
    │   │       ├── mapper/
    │   │       │   └── ProductMapper.java
    │   │       ├── dto/
    │   │       │   ├── request/...
    │   │       │   └── response/...
    │   │       └── exception/
    │   │           ├── ProductNotFoundException.java
    │   │           └── GlobalExceptionHandler.java
    │   └── resources/
    │       ├── application.properties
    │       └── logback-spring.xml
    └── test/
        ├── java/
        │   └── ro/unitbv/restlab/
        │       ├── RestLabApplicationTests.java
        │       ├── controller/
        │       │   └── ProductControllerTest.java
        │       ├── service/
        │       │   └── ProductServiceTest.java
        │       ├── repository/
        │       │   └── ProductRepositoryTest.java
        │       ├── mapper/
        │       │   └── ProductMapperTest.java
        │       ├── exception/
        │       │   ├── ProductNotFoundExceptionTest.java
        │       │   └── GlobalExceptionHandlerTest.java
        │       └── integration/
        │           ├── ProductIntegrationTest.java
        │           └── ProductTestcontainersIT.java
        └── resources/
            └── application.properties
```

Distincția este utilă în practică: putem rula rapid doar testele unitare (`mvn test`) în timpul dezvoltării, iar testele de integrare lente (`mvn verify`) doar înainte de release sau în pipeline-ul CI.

9.2. Dependențele de test în pom.xml

Se configură pom.xml pentru a include toate bibliotecile necesare. Se adăugă trei grupuri de dependențe în secțiunea `<dependencies>`:

9.2.1. spring-boot-starter-test

Această dependență aduce, ca un singur pachet, bibliotecile esențiale: JUnit 5, Mockito, AssertJ, Hamcrest, Spring Test, Spring Boot Test, JsonPath, JSONassert. Toate cu versiuni compatibile între ele, gestionate de parent POM-ul Spring Boot.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Atenție la elementul `<scope>test</scope>`: indică Maven-ului că această dependență este folosită doar în timpul testelor și nu trebuie inclusă în artefactul de producție (`.jar` final).

9.2.2. Testcontainers

Pentru testele cu containere Docker, se adăugă trei dependențe: modulul de bază pentru integrare cu JUnit 5, modulul specific pentru MySQL și suportul Spring Boot pentru `@ServiceConnection`:

```
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>mysql</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-testcontainers</artifactId>
    <scope>test</scope>
</dependency>
```

Versiunile sunt gestionate de BOM-ul Spring Boot[23], nu este necesară specificarea lor explicită.

9.2.3. Driver MySQL

Pentru ca aplicația să se poată conecta la container-ul MySQL pornit de Testcontainers, este necesar driver-ul JDBC MySQL. Se adăugă cu scope test (în viitor, dacă proiectul va folosi MySQL și în producție, dependența va fi promovată):

[23] BOM-ul în Spring Boot (de la Bill of Materials, concept preluat din inginerie și adoptat în Apache Maven) este un mecanism de gestionare centralizată a versiunilor dependențelor, prin care Spring Boot definește un set coerent și compatibil de versiuni pentru bibliotecile unui proiect.

```
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>test</scope>
</dependency>
```

9.3. Plugin-urile de build: Surefire, Failsafe, JaCoCo

Configurăm trei plugin-uri Maven în secțiunea `<build><plugins>`:

9.3.1. maven-surefire-plugin

Plugin-ul **Surefire** este responsabil pentru rularea testelor unitare în faza `test`. Este inclus implicit de `spring-boot-starter-parent`. Putem fie să nu îl declarăm deloc (configurarea implicită este suficientă), fie să îl declarăm pentru a-l personaliza:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
</plugin>
```

Surefire identifică testele după convenția numelui: `**/*Test.java`, `**/*Tests.java`, `**/Test*.java`.

9.3.2. maven-failsafe-plugin

Plugin-ul **Failsafe** este analog Surefire, dar pentru testele de integrare. Rulează în faza `verify`, separat de testele unitare. Avantajul separării este că putem rula doar testele rapide cu `mvn test`, iar suita completă cu `mvn verify`. Failsafe identifică testele după convenția `**/*IT.java`:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        <execution>
            <goals>
                <goal>integration-test</goal>
                <goal>verify</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Cele două goal-uri (`integration-test` și `verify`) trebuie ambele declarate explicit. Primul rulează testele, al doilea verifică rezultatele și eșuează build-ul dacă au fost erori.

9.3.3. jacoco-maven-plugin

Plugin-ul **JaCoCo** produce raportul de acoperire cu cod. Funcționează în două faze: prepare-agent (atașează un agent Java care instrumentează bytecode-ul în timpul rulării testelor), apoi report (procesează datele colectate și generează un raport HTML):

```
<plugin>
    <groupId>org.jacoco</groupId>
    <artifactId>jacoco-maven-plugin</artifactId>
    <version>0.8.12</version>
    <executions>
        <execution>
            <id>prepare-agent</id>
            <goals>
                <goal>prepare-agent</goal>
            </goals>
        </execution>
        <execution>
            <id>report</id>
            <phase>verify</phase>
            <goals>
                <goal>report</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

După rularea `mvn verify`, raportul este disponibil în `target/site/jacoco/index.html`. Se poate deschide în browser și se poate naviga pas cu pas: pachet → clasă → metodă → linii. Liniile colorate cu verde sunt acoperite, cele cu galben au branch-uri parțial acoperite, cele cu roșu nu sunt deloc acoperite.

9.4. application.properties pentru teste

Pentru ca testele să folosească o configurație de bază de date independentă de cea de dezvoltare/producție, creăm un fișier `src/test/resources/application.properties`. Spring Boot îl folosește automat în context-ul de test, suprascriind valorile din `src/main/resources/application.properties` (sau adăugându-le pe cele care lipsesc):

```
# Baza H2 in-memory pentru toate testele @SpringBootTest si @DataJpaTest
spring.datasource.url=jdbc:h2:mem:restlab-test;MODE=MySQL;DB_CLOSE_DELAY=-1
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=false
spring.jpa.open-in-view=false

# Console H2 nu este necesara in teste
spring.h2.console.enabled=false

# Reducem zgomotul in loguri pe durata testelor
logging.level.root=WARN
logging.level.ro.unitbv.restlab=INFO

# Endpoint-uri actuator necesare pentru testele care verifica metricile
management.endpoints.web.exposure.include=health,metrics
```

Câteva observații:

- `MODE=MySQL` determină H2 să accepte sintaxa SQL specifică MySQL – utilă dacă proiectul de producție folosește MySQL, pentru a păstra paritatea pe cât posibil;
- `DB_CLOSE_DELAY=-1` menține baza in-memory pe toată durata JVM-ului (altfel ar fi închisă imediat ce ultima conexiune se termină, înainte ca testul următor să mai poată citi);
- `ddl-auto=create-drop` recreează schema la pornire și o șterge la oprire – perfect pentru teste izolate;
- nivelul de logging redus la WARN reduce zgomotul în output-ul testelor; trecem doar pachetul aplicației la INFO pentru a vedea, totuși, cele mai importante mesaje.

Pentru testele Testcontainers, nu este nevoie de un fișier separat: adnotarea `@ServiceConnection` suprascrie automat `spring.datasource.url` cu URL-ul container-ului în timpul rulării.

10. Implementarea testelor

Trecem acum la scrierea propriu-zisă a testelor pentru fiecare clasă a aplicației. Vom proceda gradat, urcând treptele piramidei: începem cu cele mai simple teste unitare (mapper, excepție), continuăm cu testul service-ului (cu mock-uri Mockito), apoi cu slice tests (`@DataJpaTest`, `@WebMvcTest`), iar la final cu testele de integrare (`@SpringBootTest` cu H2, apoi cu Testcontainers MySQL real). [25]

10.1. Testarea ProductMapper (test unitar pur)

`ProductMapper` este clasa cea mai simplă din arhitectură: nu are dependențe, nu are stare, doar transformări pure între obiecte. Acesta este, prin excelență, cazul în care testarea unitară este trivială și totuși valoroasă: câteva minute de scris testele, ani de protecție împotriva regresiilor (de exemplu, dacă cineva uită să mapeze un nou câmp adăugat în model).

Fișier: `src/test/java/ro/unitbv/restlab/mapper/ProductMapperTest.java`

```
package ro.unitbv.restlab.mapper;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Nested;
import org.junit.jupiter.api.Test;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.model.Product;

import static org.assertj.core.api.Assertions.assertThat;

@DisplayName("ProductMapper - teste unitare pure")
class ProductMapperTest {
```

```
    private ProductMapper mapper;

    @BeforeEach
    void setUp() {
        // Mapper-ul nu are dependente; il instantiem direct,
        // fara contextul Spring. Acesta este un unit test pur.
        mapper = new ProductMapper();
    }

    @Nested
    @DisplayName("metoda toEntity()")
    class ToEntityTests {

        @Test
        @DisplayName(
            "transforma CreateProductRequest in Product cu toate campurile")
        void toEntity_shouldMapAllFields() {
            // Arrange
            CreateProductRequest request = CreateProductRequest.builder()
                    .name("Tastatura mecanica")
                    .price(349.99)
                    .stock(15)
                    .build();

            // Act
            Product product = mapper.toEntity(request);

            // Assert
            assertThat(product).isNotNull();
            assertThat(product.getName()).isEqualTo("Tastatura mecanica");
            assertThat(product.getPrice()).isEqualTo(349.99);
            assertThat(product.getStock()).isEqualTo(15);
        }


        @Test
        @DisplayName(
                "nu seteaza id-ul (ramane responsabilitatea persistentei)")
        void toEntity_shouldNotSetId() {
            CreateProductRequest request = CreateProductRequest.builder()
                    .name("Mouse").price(99.99).stock(20).build();

            Product product = mapper.toEntity(request);

            assertThat(product.getId()).isNull();
        }
    }

    @Nested   @DisplayName("metoda updateEntity()")
    class UpdateEntityTests {

        @Test
        @DisplayName("actualizeaza campurile expuse si pastreaza id-ul")
        void updateEntity_shouldKeepIdAndUpdateOtherFields() {
```

```
            Product existingProduct = Product.builder()
                    .id(42)
                    .name("Vechi").price(10.0).stock(1)
                    .build();
            UpdateProductRequest request = UpdateProductRequest.builder()
                    .name("Nou").price(20.0).stock(5)
                    .build();

            mapper.updateEntity(existingProduct, request);

            assertThat(existingProduct.getId()).isEqualTo(42);
            assertThat(existingProduct.getName()).isEqualTo("Nou");
            assertThat(existingProduct.getPrice()).isEqualTo(20.0);
            assertThat(existingProduct.getStock()).isEqualTo(5);
        }
    }

    @Nested   @DisplayName("metoda toResponse()")
    class ToResponseTests {

        @Test
        @DisplayName(
                "transforma Product in ProductResponse cu toate campurile")
        void toResponse_shouldMapAllFields() {
            Product product = Product.builder()
                    .id(7).name("Monitor").price(1299.0).stock(3)
                    .build();

            ProductResponse response = mapper.toResponse(product);

            assertThat(response).isNotNull();
            assertThat(response.getId()).isEqualTo(7);
            assertThat(response.getName()).isEqualTo("Monitor");
            assertThat(response.getPrice()).isEqualTo(1299.0);
            assertThat(response.getStock()).isEqualTo(3);
        }
    }
}
```

Câteva observații care merită subliniate:

- Folosim `@Nested` pentru a grupa testele după metoda mapper-ului. Aceasta produce un raport de execuție bine organizat: în IntelliJ sau în output-ul Maven, testele apar ierarhic pe metoda testată. Nu este obligatoriu, dar pentru clase cu mai multe metode este o practică elegantă.
- Marcăm explicit, prin comentarii, fazele Arrange-Act-Assert. Acesta este un mic gest care ajută cititorul: structura testului devine evidentă instant.
- Folosim pattern-ul Builder al Lombok-ului (`@Builder` este deja prezent în clasele `Product`, `CreateProductRequest` etc.) pentru a construi obiectele de test într-un mod fluent și citibil.
- Numele metodelor de test urmează convenția: `nume_cand_seAsteapta`. Aceasta produce nume care se citesc ca fraze: "`toEntity_shouldMapAllFields`" –

"toEntity ar trebui să mapeze toate câmpurile". `@DisplayName` completează cu o variantă în limbaj natural pentru rapoarte.

10.2. Testarea ProductNotFoundException

`ProductNotFoundException` este o clasă mică, dar care merită un test minimal pentru a documenta și verifica formatul mesajului de eroare. Mesajul este folosit în răspunsurile JSON expuse clienților, deci o modificare neintenționată ar putea afecta consumatorii API-ului.

Fișier: `src/test/java/ro/unitbv/restlab/exception/ProductNotFoundExceptionTest.java`

```
package ro.unitbv.restlab.exception;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;

@DisplayName("ProductNotFoundException - test unitar")
class ProductNotFoundExceptionTest {

    @Test
    @DisplayName("constructorul produce un mesaj care contine id-ul")
    void constructor_shouldFormatMessageWithId() {
        ProductNotFoundException ex = new ProductNotFoundException(42);

        assertThat(ex.getMessage())
                .isEqualTo("Produsul cu id-ul 42 nu a fost gasit.");
    }

    @Test
    @DisplayName(
            "este o RuntimeException - nu obliga la declarare in throws")
    void shouldBeRuntimeException() {
        ProductNotFoundException ex = new ProductNotFoundException(1);

        assertThat(ex).isInstanceOf(RuntimeException.class);
    }
}
```

Al doilea test poate părea exagerat – verifică un fapt aparent evident – dar are valoare ca documentație executabilă: orice modificare a ierarhiei excepției (de exemplu, dacă cineva o transformă în `checked exception`) va strica testul și va atrage atenția asupra impactului. Astfel de teste sunt foarte ieftine de scris și se justifică ușor.

10.3. Testarea ProductService (test unitar cu Mockito)

Service-ul este nucleul logicii de business. Are trei dependențe: `ProductRepository`, `ProductMapper` și `MeterRegistry`. Toate trei vor fi înlocuite cu test doubles, dar de tipuri diferite, pentru a ilustra diferențele [14]:

- `ProductRepository` și `ProductMapper` – vor fi mock-uri Mockito (`@Mock`); le configurăm să returneze valori cunoscute (stub) și verificăm interacțiunile (verify);
- `MeterRegistry` – va fi un fake real: `SimpleMeterRegistry`. Aceasta este o implementare reală de Micrometer, care păstrează metricile în memorie. Permite verificarea efectelor reale ale service-ului asupra metricilor (de exemplu, contorul a crescut cu 1).

Fișier: `src/test/java/ro/unitbv/restlab/service/ProductServiceTest.java`

```
package ro.unitbv.restlab.service;

import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.simple.SimpleMeterRegistry;
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.ArgumentCaptor;
import org.mockito.Mock;
import org.mockito.junit.jupiter.MockitoExtension;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.mapper.ProductMapper;
import ro.unitbv.restlab.model.Product;
import ro.unitbv.restlab.repository.ProductRepository;

import java.util.List;
import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatThrownBy;
import static org.mockito.ArgumentMatchers.*;
import static org.mockito.Mockito.*;


@ExtendWith(MockitoExtension.class)
@DisplayName("ProductService - teste unitare cu Mockito")
class ProductServiceTest {

    // Dependentele colaboratoare sunt mocks - sunt Test Doubles
    // controlate complet de test (stub + mock pentru verificari).
    @Mock private ProductRepository productRepository;
    @Mock private ProductMapper productMapper;

    // MeterRegistry-ul este folosit ca FAKE: o implementare reala,
    // dar lightweight, care functioneaza doar in memorie.
    private MeterRegistry meterRegistry;
    private ProductService productService;

    @BeforeEach
    void setUp() {
        meterRegistry = new SimpleMeterRegistry();
```

```
        // Construim manual SUT (System Under Test).
        // Avantajul injectiei prin constructor: nu avem nevoie de Spring.
        productService = new ProductService(
                productRepository, productMapper, meterRegistry);
    }

    @Nested
    @DisplayName("findAll()")
    class FindAllTests {


        @Test
        @DisplayName("cand exista produse, returneaza lista mapata")
        void findAll_whenProductsExist_shouldReturnMappedList() {
            // Arrange
            Product p1 = Product.builder().id(1).name("A")
                    .price(10.0).stock(5).build();
            Product p2 = Product.builder().id(2).name("B")
                    .price(20.0).stock(3).build();
            ProductResponse r1 = ProductResponse.builder().id(1).name("A")
                    .price(10.0).stock(5).build();
            ProductResponse r2 = ProductResponse.builder().id(2).name("B")
                    .price(20.0).stock(3).build();

            when(productRepository.findAll()).thenReturn(List.of(p1, p2));
            when(productMapper.toResponse(p1)).thenReturn(r1);
            when(productMapper.toResponse(p2)).thenReturn(r2);

            // Act
            List<ProductResponse> result = productService.findAll();

            // Assert
            assertThat(result).hasSize(2).containsExactly(r1, r2);
            verify(productRepository).findAll();
            verify(productMapper, times(2)).toResponse(any(Product.class));
        }

        @Test
        @DisplayName("cand lista este goala, returneaza lista vida")
        void findAll_whenEmpty_shouldReturnEmptyList() {
            when(productRepository.findAll()).thenReturn(List.of());

            List<ProductResponse> result = productService.findAll();

            assertThat(result).isEmpty();
            // Mapper-ul nu trebuie apelat deloc daca lista e goala
            verifyNoInteractions(productMapper);
        }
    }

    @Nested
    @DisplayName("findById()")
    class FindByIdTests {
```

```
    @Test
    @DisplayName("cand produsul exista, returneaza response-ul mapat")
    void findById_whenProductExists_shouldReturnResponse() {
        Product product = Product.builder().id(1).name("A")
                .price(10.0).stock(5).build();
        ProductResponse response = ProductResponse.builder().id(1)
                .name("A").price(10.0).stock(5).build();

        when(productRepository.findById(1)).thenReturn(Optional.of(product));
        when(productMapper.toResponse(product)).thenReturn(response);

        ProductResponse result = productService.findById(1);

        assertThat(result).isEqualTo(response);
    }

    @Test
    @DisplayName(
            "cand produsul nu exista, arunca ProductNotFoundException")
    void findById_whenProductDoesNotExist_shouldThrowException() {
       when(productRepository.findById(99)).thenReturn(Optional.empty());

        assertThatThrownBy(() -> productService.findById(99))
                .isInstanceOf(ProductNotFoundException.class)
                .hasMessageContaining("99");

        // Daca produsul nu exista, mapper-ul nu trebuie sa fie apelat
        verifyNoInteractions(productMapper);
    }
}

@Nested
@DisplayName("create()")
class CreateTests {

    @Test
    @DisplayName("salveaza produsul si returneaza response-ul")
    void create_shouldSaveAndReturnResponse() {

        CreateProductRequest request = CreateProductRequest.builder()
                .name("Nou").price(50.0).stock(10).build();
        Product toSave = Product.builder().name("Nou")
                .price(50.0).stock(10).build();
        Product saved = Product.builder().id(1).name("Nou")
                .price(50.0).stock(10).build();
        ProductResponse response = ProductResponse.builder().id(1)
                .name("Nou").price(50.0).stock(10).build();

        when(productMapper.toEntity(request)).thenReturn(toSave);
        when(productRepository.save(toSave)).thenReturn(saved);
        when(productMapper.toResponse(saved)).thenReturn(response);

        ProductResponse result = productService.create(request);

        assertThat(result).isEqualTo(response);
```

```
        verify(productRepository).save(toSave);
    }

    @Test
    @DisplayName("incrementeaza contorul products.created.count")
    void create_shouldIncrementCreatedCounter() {
        CreateProductRequest request = CreateProductRequest.builder()
                .name("X").price(10.0).stock(1).build();

        when(productMapper.toEntity(any())).thenReturn(new Product());
        when(productRepository.save(any())).thenReturn(new Product());
        when(productMapper.toResponse(any()))
            .thenReturn(new ProductResponse());

        productService.create(request);
        productService.create(request);

        // Verificam pe FAKE-ul real ca s-au inregistrat 2 incrementari
        Counter counter = meterRegistry.counter("products.created.count");
        assertThat(counter.count()).isEqualTo(2.0);
    }

    @Test
    @DisplayName(
        "trimite catre repository entitatea produsa de mapper (ArgumentCaptor)")
    void create_shouldPassMappedEntityToRepository() {
        CreateProductRequest request = CreateProductRequest.builder()
                .name("X").price(10.0).stock(5).build();
        Product mapped = Product.builder().name("X")
                .price(10.0).stock(5).build();

        when(productMapper.toEntity(request)).thenReturn(mapped);
        when(productRepository.save(any(Product.class)))
                .thenReturn(Product.builder().id(1).name("X")
                        .price(10.0).stock(5).build());
        when(productMapper.toResponse(any()))
                .thenReturn(ProductResponse.builder().id(1).build());

        productService.create(request);

        // ArgumentCaptor permite inspectarea argumentelor primite
        // efectiv de catre o metoda mockata.
        ArgumentCaptor<Product> captor =
                ArgumentCaptor.forClass(Product.class);
        verify(productRepository).save(captor.capture());

        Product captured = captor.getValue();
        assertThat(captured.getName()).isEqualTo("X");
        assertThat(captured.getPrice()).isEqualTo(10.0);
        assertThat(captured.getStock()).isEqualTo(5);
    }
}
```

```
    @Nested
    @DisplayName("update()")
    class UpdateTests {

        @Test
        @DisplayName(
                "cand produsul exista, il actualizeaza si il returneaza")
        void update_whenProductExists_shouldUpdateAndReturn() {
            Product existing = Product.builder().id(1).name("Vechi")
                    .price(10.0).stock(2).build();
            UpdateProductRequest req = UpdateProductRequest.builder()
                    .name("Nou").price(20.0).stock(5).build();
            Product updated = Product.builder().id(1).name("Nou")
                    .price(20.0).stock(5).build();
            ProductResponse response = ProductResponse.builder().id(1)
                    .name("Nou").price(20.0).stock(5).build();

            when(productRepository.findById(1)).thenReturn(Optional.of(existing));
            when(productRepository.save(existing)).thenReturn(updated);
            when(productMapper.toResponse(updated)).thenReturn(response);

            ProductResponse result = productService.update(1, req);

            assertThat(result).isEqualTo(response);
            // Verificam ca service-ul a delegat actualizarea catre mapper
            verify(productMapper).updateEntity(existing, req);
        }

        @Test
        @DisplayName(
              "cand produsul nu exista, arunca exceptie si NU apeleaza save")
        void update_whenProductDoesNotExist_shouldThrowAndNotSave() {
            UpdateProductRequest req = UpdateProductRequest.builder()
                    .name("Y").price(1.0).stock(1).build();
            when(productRepository.findById(999)).thenReturn(Optional.empty());

            assertThatThrownBy(() -> productService.update(999, req))
                    .isInstanceOf(ProductNotFoundException.class)
                    .hasMessageContaining("999");

            verify(productRepository, never()).save(any());
            verifyNoInteractions(productMapper);
        }
    }

    @Nested
    @DisplayName("deleteById()")
    class DeleteByIdTests {

        @Test
        @DisplayName("cand produsul exista, il sterge")
        void deleteById_whenExists_shouldDelete() {
            when(productRepository.existsById(1)).thenReturn(true);

            productService.deleteById(1);
```

```
            verify(productRepository).deleteById(1);
        }

        @Test
        @DisplayName(
            "cand produsul nu exista, arunca exceptie si NU apeleaza delete")
        void deleteById_whenNotExists_shouldThrowAndNotDelete() {
            when(productRepository.existsById(999)).thenReturn(false);

            assertThatThrownBy(() -> productService.deleteById(999))
                    .isInstanceOf(ProductNotFoundException.class)
                    .hasMessageContaining("999");

            verify(productRepository, never()).deleteById(anyInt());
        }
    }
}
```

Câteva tehnici importante ilustrate de acest test merită discutate explicit:

10.3.1. Combinarea de mock-uri și fake-uri

Folosim deliberat tipuri diferite de test doubles. Repository-ul și mapper-ul sunt mock-uri (Mockito) – nu am vrea ca testul să atingă o bază reală sau să ruleze logica reală a mapper-ului (care, oricum, este testată separat). Pentru `MeterRegistry`, însă, mock-uirea ar fi nepractică: API-ul Micrometer folosește un pattern de builder fluid (`Counter.builder().tag(...).register(meterRegistry).increment()`), care ar produce mock-uri imbricate fragile. Folosirea `SimpleMeterRegistry` (un fake) este, în acest caz, mai simplă și mai robustă: API-ul Micrometer rulează real, iar la final inspectăm starea registrului direct prin `meterRegistry.counter("name").count()`.

10.3.2. ArgumentCaptor

Atunci când un mock primește un argument construit dinamic în interiorul SUT-ului (în cazul nostru, entitatea `Product` produsă de mapper), nu putem să verificăm argumentul direct cu `verify(repo).save(produsAsteptat)` – pentru că nu cunoaștem instanța exactă creată intern. Soluția este `ArgumentCaptor`: îl atașăm la `verify`, capturăm argumentul efectiv primit, și apoi facem aserțiuni asupra lui.

10.3.3. Verificări negative

Testele de eroare verifică nu doar că s-a aruncat excepția corectă, ci și că efectele secundare nepermise nu s-au produs. De exemplu, în `update_whenProductDoesNotExist_shouldThrowAndNotSave`, verificăm că `save` nu a fost apelat (`verify(repo, never()).save(any())`). Această verificare are valoare reală: dacă cineva ar refactoriza service-ul și ar muta apelul `save` înainte de verificarea existenței, testul ar prinde imediat regresia.

10.3.4. verifyNoInteractions

`verifyNoInteractions(mock)` verifică faptul că un mock nu a primit niciun apel pe nicio metodă. Este o aserțiune puternică: dacă cineva adaugă, în viitor, o invocare neașteptată a mapper-ului în calea de eroare, testul va eșua. Acest tip de verificare ajută la menținerea contractului implicit dintre componente.

10.4. Testarea GlobalExceptionHandler

GlobalExceptionHandler convertește excepțiile în răspunsuri HTTP cu structură uniformă. Are dependențe (`MeterRegistry`) și interacționează cu obiecte ale infrastructurii web (`HttpServletRequest`, `MethodArgumentNotValidException`). Ne interesează să-l testăm direct, ca pe o clasă obișnuită, fără a porni contextul Spring: îl instanțiem manual, mock-uim cele câteva dependențe pe care le folosește în metode și verificăm răspunsurile produse.

Fișier: `src/test/java/ro/unitbv/restlab/exception/GlobalExceptionHandlerTest.java`

```
package ro.unitbv.restlab.exception;

import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.simple.SimpleMeterRegistry;
import jakarta.servlet.http.HttpServletRequest;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.validation.FieldError;
import org.springframework.web.bind.MethodArgumentNotValidException;
import ro.unitbv.restlab.dto.response.ErrorResponse;
import ro.unitbv.restlab.dto.response.ValidationExceptionResponse;

import java.util.List;


import static org.assertj.core.api.Assertions.assertThat;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

@DisplayName("GlobalExceptionHandler - teste unitare")
class GlobalExceptionHandlerTest {

    private MeterRegistry meterRegistry;
    private GlobalExceptionHandler handler;
    private HttpServletRequest request;

    @BeforeEach
    void setUp() {
        meterRegistry = new SimpleMeterRegistry();
        handler = new GlobalExceptionHandler(meterRegistry);
```

```
        // HttpServletRequest este o interfata; o folosim ca STUB
        // returnand mereu acelasi URI fara o implementare reala de servlet.
        request = mock(HttpServletRequest.class);
        when(request.getRequestURI()).thenReturn("/api/products/99");
    }

    @Test
    @DisplayName(
       "handleProductNotFound - returneaza 404 cu structura ErrorResponse completa")
    void handleProductNotFound_shouldReturn404() {
        ProductNotFoundException ex = new ProductNotFoundException(99);

        ResponseEntity<ErrorResponse> response =
                handler.handleProductNotFound(ex, request);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
        assertThat(response.getBody()).isNotNull();
        assertThat(response.getBody().getStatus()).isEqualTo(404);
        assertThat(response.getBody().getError()).isEqualTo("Not Found");
        assertThat(response.getBody().getMessage()).contains("99");
        assertThat(response.getBody().getPath()).isEqualTo("/api/products/99");
        assertThat(response.getBody().getTimestamp()).isNotNull();
    }

    @Test
    @DisplayName(
    "handleProductNotFound - incrementeaza contorul products.notfound.count")
    void handleProductNotFound_shouldIncrementCounter() {
        ProductNotFoundException ex = new ProductNotFoundException(99);

        handler.handleProductNotFound(ex, request);
        handler.handleProductNotFound(ex, request);

        Counter counter = meterRegistry.counter("products.notfound.count");
        assertThat(counter.count()).isEqualTo(2.0);
    }


    @Test
    @DisplayName(
            "handleValidationException - returneaza 400 cu erori per camp")
    void handleValidationException_shouldReturn400WithFieldErrors() {
        // Construim o BindingResult mock-uita care raporteaza
        // doua erori de validare: una pe 'name', una pe 'price'.
        BindingResult bindingResult = mock(BindingResult.class);
        FieldError nameError = new FieldError(
                "createProductRequest", "name", "obligatoriu");
        FieldError priceError = new FieldError(
                "createProductRequest", "price", "trebuie > 0");
        when(bindingResult.getFieldErrors())
                .thenReturn(List.of(nameError, priceError));

        MethodArgumentNotValidException ex =
                mock(MethodArgumentNotValidException.class);
```

```
        when(ex.getBindingResult()).thenReturn(bindingResult);

        ResponseEntity<ValidationExceptionResponse> response =
                handler.handleValidationException(ex, request);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
        assertThat(response.getBody()).isNotNull();
        assertThat(response.getBody().getStatus()).isEqualTo(400);
        assertThat(response.getBody().getValidationErrors())
                .containsEntry("name", "obligatoriu")
                .containsEntry("price", "trebuie > 0");
        assertThat(response.getBody().getPath()).isEqualTo("/api/products/99");
    }

    @Test
    @DisplayName(
  "handleGenericException - returneaza 500 cu mesaj generic (fara expunere interna)")
    void handleGenericException_shouldReturn500() {
        Exception ex = new RuntimeException("ceva neasteptat intern");

        ResponseEntity<ErrorResponse> response =
                handler.handleGenericException(ex, request);

        assertThat(response.getStatusCode())
                .isEqualTo(HttpStatus.INTERNAL_SERVER_ERROR);
        assertThat(response.getBody().getStatus()).isEqualTo(500);
        // Mesajul real al exceptiei nu trebuie expus catre client
        assertThat(response.getBody().getMessage())
                .isEqualTo("A aparut o eroare interna.")
                .doesNotContain("ceva neasteptat intern");
    }
}
```

Ultima aserțiune merită atenție specială. Verificăm că mesajul intern al excepției nu apare în răspunsul către client. Este o verificare de securitate: expunerea de detalii interne (mesaje de eroare, stack traces, queryuri SQL) către clienți este o vulnerabilitate frecventă în aplicațiile web. Testul nostru documentează intenția de a NU expune asemenea informații și o impune în mod automat: orice modificare neglijentă va fi prinsă imediat.

10.5. Testarea ProductRepository (slice test cu @DataJpaTest)

Pentru repository, folosim slice-ul `@DataJpaTest`. Acesta pornește un context Spring minimal: doar configurația JPA, un DataSource embedded (H2 implicit), EntityManager și repository-urile Spring Data. Fiecare test rulează într-o tranzacție cu rollback automat la final, deci testele sunt complet izolate fără să fie nevoie de curățare manuală. Beneficiul colateral este viteza: pornirea acestui slice ia, tipic, sub o secundă. [1, 5]

Fișier: `src/test/java/ro/unitbv/restlab/repository/ProductRepositoryTest.java`

```
package ro.unitbv.restlab.repository;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.orm.jpa.DataJpaTest;
import org.springframework.boot.test.autoconfigure.orm.jpa.TestEntityManager;
import ro.unitbv.restlab.model.Product;

import java.util.List;
import java.util.Optional;

import static org.assertj.core.api.Assertions.assertThat;

@DataJpaTest
@DisplayName(
       "ProductRepository - slice test cu @DataJpaTest si TestEntityManager")
class ProductRepositoryTest {

    @Autowired private TestEntityManager entityManager;
    @Autowired private ProductRepository productRepository;

    @Test
    @DisplayName("save - persista entitatea si genereaza id automat")
    void save_shouldPersistAndGenerateId() {
        Product product = Product.builder()
                .name("Tastatura").price(199.99).stock(10)
                .build();

        Product saved = productRepository.save(product);

        assertThat(saved.getId()).isNotNull();
        assertThat(saved.getName()).isEqualTo("Tastatura");


        // Validam ca persistenta s-a propagat la nivelul EntityManager-ului
        Product found = entityManager.find(Product.class, saved.getId());
        assertThat(found).isNotNull();
        assertThat(found.getName()).isEqualTo("Tastatura");
    }

    @Test
    @DisplayName("findById - gaseste un produs persistat anterior")
    void findById_shouldReturnPersistedProduct() {
        Product persisted = entityManager.persistAndFlush(
                Product.builder().name("Mouse")
                        .price(49.0).stock(20).build());

        Optional<Product> result = productRepository.findById(persisted.getId());

        assertThat(result).isPresent();
        assertThat(result.get().getName()).isEqualTo("Mouse");
    }

    @Test
    @DisplayName("findById - returneaza Optional vid pentru id inexistent")
    void findById_whenNotFound_shouldReturnEmpty() {
        Optional<Product> result = productRepository.findById(99999);
```

```
        assertThat(result).isEmpty();
    }

    @Test
    @DisplayName("findAll - returneaza toate produsele persistate")
    void findAll_shouldReturnAllPersisted() {
        entityManager.persist(Product.builder()
                .name("A").price(1.0).stock(1).build());
        entityManager.persist(Product.builder()
                .name("B").price(2.0).stock(2).build());
        entityManager.persist(Product.builder()
                .name("C").price(3.0).stock(3).build());
        entityManager.flush();

        List<Product> result = productRepository.findAll();

        assertThat(result).hasSize(3);
        assertThat(result).extracting(Product::getName)
                .containsExactlyInAnyOrder("A", "B", "C");
    }

    @Test
    @DisplayName("deleteById - sterge produsul din baza de date")
    void deleteById_shouldRemoveFromDatabase() {
        Product persisted = entityManager.persistAndFlush(
                Product.builder().name("X")
                        .price(1.0).stock(1).build());
        Integer id = persisted.getId();

        productRepository.deleteById(id);
        entityManager.flush();

        assertThat(entityManager.find(Product.class, id)).isNull();
    }

    @Test
    @DisplayName(
            "existsById - returneaza true pentru id existent, false altfel")
    void existsById_shouldWorkCorrectly() {
        Product persisted = entityManager.persistAndFlush(
                Product.builder().name("Y")
                        .price(1.0).stock(1).build());

        assertThat(productRepository.existsById(persisted.getId())).isTrue();
        assertThat(productRepository.existsById(99999)).isFalse();
    }

    @Test
    @DisplayName("count - reflecta corect numarul de entitati persistate")
    void count_shouldReflectPersistedEntities() {
        assertThat(productRepository.count()).isZero();

        entityManager.persist(Product.builder()
                .name("A").price(1.0).stock(1).build());
        entityManager.persist(Product.builder()
```

```
                .name("B").price(2.0).stock(2).build());
        entityManager.flush();

        assertThat(productRepository.count()).isEqualTo(2);
    }
}
```

Câteva idei importante:

- `TestEntityManager` este un wrapper subțire peste EntityManager-ul JPA, oferit special pentru teste. Avantajul lui față de a folosi direct `ProductRepository` pentru pregătirea datelor este izolarea: dacă există un bug în repository, fixture-urile nu sunt afectate, iar testul are sens.
- `persistAndFlush` este folosit deliberat: persistă entitatea ȘI forțează flush-ul către baza de date, asigurându-se că orice constrângere SQL ar fi violată va eșua imediat (și nu doar la finalul tranzacției).
- Pentru un repository extins cu metode personalizate (de exemplu, derivate de queryuri sau `@Query`), aici am scrie testele lor – exact pe acest slice, fără a porni controllere sau servicii.

10.6. Testarea ProductController (slice test cu @WebMvcTest)

Pentru controller, folosim `@WebMvcTest(ProductController.class)`. Acesta pornește doar stratul web: serializatorii Jackson, infrastructura de validare Bean Validation, MockMvc, controller-ul indicat și orice `@RestControllerAdvice` disponibil. Service-ul este înlocuit cu un mock prin `@MockBean`, iar dependențele sale tranzitive (repository, datasource) nu sunt nici măcar instanțiate. [1, 2]

Există o subtilitate: `@WebMvcTest` încarcă automat și `GlobalExceptionHandler` (care este `@RestControllerAdvice`), iar acesta cere un bean `MeterRegistry`. Trebuie să-l furnizăm. Cea mai curată soluție este o clasă internă cu adnotarea `@TestConfiguration` care expune un `SimpleMeterRegistry` ca bean.

Fișier: `src/test/java/ro/unitbv/restlab/controller/ProductControllerTest.java`

```
package ro.unitbv.restlab.controller;

import com.fasterxml.jackson.databind.ObjectMapper;
import io.micrometer.core.instrument.MeterRegistry;
import io.micrometer.core.instrument.simple.SimpleMeterRegistry;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.boot.test.mock.mockito.MockBean;
import org.springframework.context.annotation.Bean;
import org.springframework.http.MediaType;
import org.springframework.test.web.servlet.MockMvc;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
```

```
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.exception.ProductNotFoundException;
import ro.unitbv.restlab.service.ProductService;

import java.util.List;

import static org.hamcrest.Matchers.containsString;
import static org.hamcrest.Matchers.hasSize;
import static org.mockito.ArgumentMatchers.any;
import static org.mockito.ArgumentMatchers.eq;
import static org.mockito.Mockito.*;
import static
org.springframework.test.web.servlet.request.MockMvcRequestBuilders.*;
import static
org.springframework.test.web.servlet.result.MockMvcResultMatchers.*;

@WebMvcTest(ProductController.class)
@DisplayName("ProductController - slice test cu @WebMvcTest si MockMvc")
class ProductControllerTest {

    @Autowired private MockMvc mockMvc;
    @Autowired private ObjectMapper objectMapper;

    // @MockBean inlocuieste bean-ul corespunzator din contextul Spring
    // de test cu un mock Mockito.
    @MockBean private ProductService productService;

    /**
     * GlobalExceptionHandler este auto-incarcat de @WebMvcTest pentru
     * a procesa erorile, iar el cere un MeterRegistry. Furnizam o
     * implementare reala lightweight (FAKE) prin aceasta TestConfiguration.
     */

    @TestConfiguration
    static class TestConfig {
        @Bean
        MeterRegistry meterRegistry() {
            return new SimpleMeterRegistry();
        }
    }

    // ====================== GET /api/products ====================

    @Test
    @DisplayName("GET /api/products - returneaza lista produselor cu 200 OK")
    void getAllProducts_shouldReturnProductList() throws Exception {
        ProductResponse p1 = ProductResponse.builder()
                .id(1).name("A").price(10.0).stock(5).build();
        ProductResponse p2 = ProductResponse.builder()
                .id(2).name("B").price(20.0).stock(3).build();
        when(productService.findAll()).thenReturn(List.of(p1, p2));

        mockMvc.perform(get("/api/products"))
                .andExpect(status().isOk())
```

```
                .andExpect(jsonPath("$", hasSize(2)))
                .andExpect(jsonPath("$[0].id").value(1))
                .andExpect(jsonPath("$[0].name").value("A"))
                .andExpect(jsonPath("$[1].id").value(2));
    }

    // ================== GET /api/products/{id} ==================

    @Test
    @DisplayName("GET /api/products/{id} - cand exista, returneaza 200 OK")
    void getProductById_whenExists_shouldReturn200() throws Exception {
        ProductResponse response = ProductResponse.builder()
                .id(1).name("A").price(10.0).stock(5).build();
        when(productService.findById(1)).thenReturn(response);

        mockMvc.perform(get("/api/products/1"))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.name").value("A"));
    }

    @Test
    @DisplayName(
         "GET /api/products/{id} - cand nu exista, returneaza 404 cu ErrorResponse")
    void getProductById_whenNotFound_shouldReturn404() throws Exception {
        when(productService.findById(99))
                .thenThrow(new ProductNotFoundException(99));

        mockMvc.perform(get("/api/products/99"))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.status").value(404))
                .andExpect(jsonPath("$.error").value("Not Found"))
                .andExpect(jsonPath("$.message", containsString("99")))
                .andExpect(jsonPath("$.path").value("/api/products/99"));
    }

    // ===================== POST /api/products =====================

    @Test
    @DisplayName(
      "POST /api/products - cand request-ul e valid, returneaza 201 Created")
    void createProduct_whenValid_shouldReturn201() throws Exception {
        CreateProductRequest request = CreateProductRequest.builder()
                .name("Nou").price(50.0).stock(10).build();
        ProductResponse response = ProductResponse.builder()
                .id(1).name("Nou").price(50.0).stock(10).build();

        when(productService.create(any(CreateProductRequest.class)))
                .thenReturn(response);

        mockMvc.perform(post("/api/products")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isCreated())
                .andExpect(jsonPath("$.id").value(1))
```

```
                .andExpect(jsonPath("$.name").value("Nou"));
    }

    @Test
    @DisplayName(
      "POST /api/products - cand numele e gol, returneaza 400 cu eroare de validare")
    void createProduct_whenNameIsBlank_shouldReturn400() throws Exception {
        CreateProductRequest request = CreateProductRequest.builder()
                .name("").price(50.0).stock(10).build();

        mockMvc.perform(post("/api/products")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.status").value(400))
                .andExpect(jsonPath("$.validationErrors.name").exists());
    }

    @Test
    @DisplayName(
     "POST /api/products - cand pretul e 0 (sub limita @DecimalMin), returneaza 400")
    void createProduct_whenPriceIsZero_shouldReturn400() throws Exception {
        CreateProductRequest request = CreateProductRequest.builder()
                .name("X").price(0.0).stock(10).build();

        mockMvc.perform(post("/api/products")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.validationErrors.price").exists());
    }

    @Test
    @DisplayName(
            "POST /api/products - cand stocul e negativ, returneaza 400")
    void createProduct_whenStockIsNegative_shouldReturn400()
                                                        throws Exception {
        CreateProductRequest request = CreateProductRequest.builder()
                .name("X").price(50.0).stock(-1).build();

        mockMvc.perform(post("/api/products")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.validationErrors.stock").exists());
    }

    @Test
    @DisplayName(
         "POST /api/products - cand numele depaseste 100 caractere, returneaza 400")
    void createProduct_whenNameTooLong_shouldReturn400() throws Exception {
        String longName = "A".repeat(101);
        CreateProductRequest request = CreateProductRequest.builder()
                .name(longName).price(50.0).stock(10).build();
```

```
        mockMvc.perform(post("/api/products")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
                .andExpect(jsonPath("$.validationErrors.name").exists());
    }

    // ================== PUT /api/products/{id} ==================
    @Test
    @DisplayName(
     "PUT /api/products/{id} - cand exista, returneaza 200 OK cu produsul actualizat")
    void updateProduct_whenExists_shouldReturn200() throws Exception {

        UpdateProductRequest request = UpdateProductRequest.builder()
                .name("Actualizat").price(75.0).stock(20).build();
        ProductResponse response = ProductResponse.builder()
                .id(1).name("Actualizat").price(75.0).stock(20).build();
        when(productService.update(eq(1), any(UpdateProductRequest.class)))
                .thenReturn(response);

        mockMvc.perform(put("/api/products/1")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.id").value(1))
                .andExpect(jsonPath("$.name").value("Actualizat"));
    }

    @Test
    @DisplayName("PUT /api/products/{id} - cand nu exista, returneaza 404")
    void updateProduct_whenNotFound_shouldReturn404() throws Exception {

        UpdateProductRequest request = UpdateProductRequest.builder()
                .name("X").price(75.0).stock(20).build();
        when(productService.update(eq(99), any(UpdateProductRequest.class)))
                .thenThrow(new ProductNotFoundException(99));

        mockMvc.perform(put("/api/products/99")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.status").value(404));
    }

    @Test
    @DisplayName("PUT /api/products/{id} - cu request invalid, returneaza 400")
    void updateProduct_whenInvalid_shouldReturn400() throws Exception {

        UpdateProductRequest request = UpdateProductRequest.builder()
                .name("").price(-1.0).stock(-5).build();

        mockMvc.perform(put("/api/products/1")
                        .contentType(MediaType.APPLICATION_JSON)
                        .content(objectMapper.writeValueAsString(request)))
                .andExpect(status().isBadRequest())
```

```
                .andExpect(jsonPath("$.validationErrors.name").exists())
                .andExpect(jsonPath("$.validationErrors.price").exists())
                .andExpect(jsonPath("$.validationErrors.stock").exists());
    }

    // ================= DELETE /api/products/{id} =================

    @Test
    @DisplayName(
        "DELETE /api/products/{id} - cand exista, returneaza 204 No Content")
    void deleteProduct_whenExists_shouldReturn204() throws Exception {
        doNothing().when(productService).deleteById(1);

        mockMvc.perform(delete("/api/products/1"))
                .andExpect(status().isNoContent());

        verify(productService).deleteById(1);
    }

    @Test
    @DisplayName(
            "DELETE /api/products/{id} - cand nu exista, returneaza 404")
    void deleteProduct_whenNotFound_shouldReturn404() throws Exception {
        doThrow(new ProductNotFoundException(99))
                .when(productService).deleteById(99);

        mockMvc.perform(delete("/api/products/99"))
                .andExpect(status().isNotFound())
                .andExpect(jsonPath("$.status").value(404));
    }
}
```

Câteva idei merită evidențiate:

10.6.1. Testele de validare

Testele care verifică validarea (`createProduct_whenNameIsBlank`, `createProduct_whenPriceIsZero`, etc.) nu apelează service-ul mock-uit; nu este nevoie să configurăm vreun `when()` pe el. Validarea se petrece înainte ca request-ul să ajungă la controller (în stratul Bean Validation). Dacă request-ul este invalid, Spring aruncă `MethodArgumentNotValidException` care este interceptată de `GlobalExceptionHandler`, iar răspunsul este 400 cu structura `ValidationExceptionResponse`. Testele acoperă toate regulile de validare definite pe DTO-uri (`@NotBlank`, `@DecimalMin`, `@Min`, `@Size`).

10.6.2. Aserțiunile JSONPath

Aserțiunile pe răspunsul JSON folosesc expresii JSONPath. Exemple folosite: `$` = rădăcina, `$.id` = câmpul `id`, `$[0].name` = numele primului element din array, `$.validationErrors.price` = câmpul `price` din obiectul `validationErrors`. `hasSize(2)` (Hamcrest matcher) verifică dimensiunea unui array, `containsString("99")` verifică o substring, `exists()` doar prezența unui câmp.

10.6.3. Mock-uirea metodelor void

Pentru metode void (cum ar fi `deleteById`), Mockito nu poate folosi sintaxa `when().thenReturn()` (nu există valoare returnată). Sintaxa alternativă este `doNothing().when(mock).method()` sau `doThrow(...).when(mock).method()`. Această diferență de sintaxă este una dintre subtilitățile Mockito care apar frecvent.

10.7. Test de integrare end-to-end (@SpringBootTest cu H2)

După ce am acoperit fiecare strat în izolare, urmează verificarea că aceștia funcționează corect împreună. Aceasta este sarcina testului de integrare end-to-end. Pornim contextul Spring complet, lăsăm Spring Boot să configureze H2-ul in-memory pe baza application.properties din `src/test/resources`, expunem aplicația pe un port aleator și interacționăm cu ea ca orice alt client HTTP, prin `TestRestTemplate`. [1]

Fișier: `src/test/java/ro/unitbv/restlab/integration/ProductIntegrationTest.java`

```
package ro.unitbv.restlab.integration;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.request.UpdateProductRequest;
import ro.unitbv.restlab.dto.response.ErrorResponse;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.dto.response.ValidationExceptionResponse;
import ro.unitbv.restlab.repository.ProductRepository;

import static org.assertj.core.api.Assertions.assertThat;

/**
 * Test de integrare la nivelul intregii aplicatii: porneste contextul
 * Spring Boot complet pe un port aleator, foloseste o baza H2 in-memory
 * si interactioneaza cu aplicatia prin HTTP real (TestRestTemplate).
 *
 * Acest test exerseaza intregul flux: rutare HTTP -> deserializare ->
 * validare -> controller -> service -> repository -> H2 -> serializare
 * raspuns -> client.
 */
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@DisplayName("Test de integrare end-to-end - flux complet CRUD")
class ProductIntegrationTest {

    @Autowired private TestRestTemplate restTemplate;
    @Autowired private ProductRepository productRepository;
```

```
@BeforeEach
void setUp() {
    // Curatam baza inainte de fiecare test pentru izolare.
    productRepository.deleteAll();
}

@Test
@DisplayName("flux CRUD complet: create -> read -> update -> delete")
void fullCrudFlow_shouldWorkEndToEnd() {
    // ========== 1. CREATE ==========
    CreateProductRequest createRequest = CreateProductRequest.builder()
            .name("Tastatura").price(299.99).stock(15)
            .build();

    ResponseEntity<ProductResponse> createResp =
                                    restTemplate.postForEntity(
                                    "/api/products",
                                    createRequest,
                                    ProductResponse.class);

    assertThat(createResp.getStatusCode()).isEqualTo(HttpStatus.CREATED);
    assertThat(createResp.getBody()).isNotNull();
    Integer id = createResp.getBody().getId();
    assertThat(id).isNotNull();
    assertThat(createResp.getBody().getName()).isEqualTo("Tastatura");

    // ========== 2. READ ==========
    ResponseEntity<ProductResponse> readResp = restTemplate.getForEntity(
            "/api/products/" + id, ProductResponse.class);
    assertThat(readResp.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(readResp.getBody().getName()).isEqualTo("Tastatura");

    // ========== 3. UPDATE ==========
    UpdateProductRequest updateRequest = UpdateProductRequest.builder()
            .name("Tastatura mecanica").price(399.99).stock(10)
            .build();
    ResponseEntity<ProductResponse> updateResp = restTemplate.exchange(
            "/api/products/" + id, HttpMethod.PUT,
            new HttpEntity<>(updateRequest), ProductResponse.class);
    assertThat(updateResp.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(updateResp.getBody().getName())
            .isEqualTo("Tastatura mecanica");

    // ========== 4. DELETE ==========
    ResponseEntity<Void> deleteResp = restTemplate.exchange(
            "/api/products/" + id, HttpMethod.DELETE, null, Void.class);
    assertThat(deleteResp.getStatusCode())
            .isEqualTo(HttpStatus.NO_CONTENT);

    // ========== 5. READ dupa DELETE - asteptam 404 ==========
    ResponseEntity<ErrorResponse> readAfterDelete =
            restTemplate.getForEntity(
                    "/api/products/" + id, ErrorResponse.class);
    assertThat(readAfterDelete.getStatusCode())
```

```
                .isEqualTo(HttpStatus.NOT_FOUND);
    }

    @Test
    @DisplayName("POST cu date invalide returneaza 400 si NU persista nimic")
    void postWithInvalidData_shouldReturn400AndNotPersist() {
        CreateProductRequest invalid = CreateProductRequest.builder()
                .name("").price(-10).stock(-5).build();

        ResponseEntity<ValidationExceptionResponse> response =
                restTemplate.postForEntity(
                        "/api/products",
                        invalid,
                        ValidationExceptionResponse.class);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.BAD_REQUEST);
        assertThat(response.getBody().getValidationErrors())
                .containsKeys("name", "price", "stock");

        // Verificare suplimentara la nivelul bazei de date:
        // request-ul invalid nu a creat nimic.
        assertThat(productRepository.count()).isZero();
    }


    @Test
    @DisplayName(
        "GET pentru id inexistent returneaza 404 cu structura ErrorResponse")
    void getNonExistentId_shouldReturn404WithErrorBody() {
        ResponseEntity<ErrorResponse> response = restTemplate.getForEntity(
                "/api/products/12345", ErrorResponse.class);

        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
        assertThat(response.getBody().getStatus()).isEqualTo(404);
        assertThat(response.getBody().getMessage()).contains("12345");
        assertThat(response.getBody().getPath()).isEqualTo("/api/products/12345");
    }
}
```

Câteva detalii importante:

- `WebEnvironment.RANDOM_PORT` pornește serverul Tomcat embedded pe un port aleator. Spring Boot injectează automat un TestRestTemplate configurat cu URL-ul corect, deci în teste putem folosi căi relative (`"/api/products"`) fără să ne preocupe portul exact.
- Curățarea bazei de date la `@BeforeEach` (`productRepository.deleteAll()`) garantează izolarea: fiecare test pornește de la o stare cunoscută. Spre deosebire de `@DataJpaTest`, `@SpringBootTest` nu rulează automat în tranzacții cu rollback, deci avem nevoie de curățare manuală.
- Pentru `DELETE` și `PUT` folosim `restTemplate.exchange()` pentru că metodele `restTemplate.delete()` și `restTemplate.put()` sunt void și nu permit citirea statusului HTTP din răspuns.

- Verificăm și asupra REPOSITORY-ului că date invalide nu au persistat (`productRepository.count()).isZero()`: aceasta este o asertiune suplimentară care prinde o categorie întreagă de bug-uri (de exemplu, un service care salvează înainte de validare).

10.8. Smoke test (@SpringBootTest minimal)

Spring Initializr generează automat o clasă de test cu numele `RestLabApplicationTests` și o singură metodă, `contextLoads`. Această metodă pare goală, dar are o valoare reală: dacă trece, înseamnă că aplicația poate să pornească. Defectele pe care le prinde acest test:

- anotare incorectă (de exemplu, o clasă care extinde o interfață greșit);
- dependență care lipsește de pe classpath (un bean cere altceva);
- configurare incorectă (un property obligatoriu lipsește din application.properties);
- un fișier SQL de inițializare cu sintaxă greșită;
- ciclicitate în beans (A depinde de B, B depinde de A).

Fișier: `src/test/java/ro/unitbv/restlab/RestLabApplicationTests.java`

```
package ro.unitbv.restlab;

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;


@SpringBootTest
@DisplayName("Smoke test - contextul Spring se incarca corect")
class RestLabApplicationTests {

    /**
     * Smoke test minimal: daca acest test trece, inseamna ca:
     *   - toate clasele @Component, @Service, @Repository, @RestController
     *     au fost descoperite si instantiate cu succes;
     *   - dependentele au fost rezolvate corect prin DI;
     *   - configuratiile (application.properties) au fost aplicate;
     *   - schema H2 s-a creat fara erori (ddl-auto=create);
     *   - actuator-ul si registry-ul de metrici au pornit.
     */
    @Test
    @DisplayName("contextLoads - aplicatia porneste fara erori")
    void contextLoads() {
        // Nu este nevoie de cod; @SpringBootTest porneste contextul.
    }
}
```

Acest test este, deliberat, foarte general – verifică o suprafață mare de configurare cu un cost de scriere zero. Este smoke test-ul arhetipal: dacă produce fum, înseamnă că ceva fundamental este greșit.

10.9. Test de integrare cu Testcontainers (MySQL real)

În plus față de testul de integrare cu H2, vom scrie unul cu MySQL real prin Testcontainers. Acesta va rula într-o fază separată a build-ului (`verify` prin Maven Failsafe), deci nu va încetini ciclul rapid de dezvoltare.

Numele clasei se termină în `IT`, nu în `Test`, pentru ca plugin-ul Maven Failsafe să-l ruleze (în loc de Surefire). Conventia este o consecință directă a configurației de plugin-uri stabilită în secțiunea anterioară.

Fișier: `src/test/java/ro/unitbv/restlab/integration/ProductTestcontainersIT.java`

```
package ro.unitbv.restlab.integration;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.client.TestRestTemplate;
import org.springframework.boot.testcontainers.service.connection.ServiceConnection;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.testcontainers.containers.MySQLContainer;
import org.testcontainers.junit.jupiter.Container;
import org.testcontainers.junit.jupiter.Testcontainers;
import ro.unitbv.restlab.dto.request.CreateProductRequest;
import ro.unitbv.restlab.dto.response.ProductResponse;
import ro.unitbv.restlab.repository.ProductRepository;

import static org.assertj.core.api.Assertions.assertThat;

/**
 * Test de integrare ce porneste o instanta REALA de MySQL intr-un container
 * Docker prin Testcontainers. Diferenta fata de @SpringBootTest cu H2:
 *   - rulam impotriva aceluiasi engine ca in productie ->
incompatibilitatile
 *     dialectale sunt prinse aici;
 *   - este ceva mai lent (porneste un container);
 *   - este complet izolat de mediul de dezvoltare al programatorului.
 *
 * Conventia *IT (Integration Test) la finalul numelui clasei face ca
 * acest test sa fie rulat de plugin-ul maven-failsafe-plugin in faza
 * "verify", separat de testele unitare rulate de Surefire in faza "test".
 */
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@Testcontainers
@DisplayName("Test de integrare cu Testcontainers (MySQL real prin Docker)")
class ProductTestcontainersIT {

    /**
     * @Container marcheaza un container care va fi pornit/oprit automat
     * de catre extensia JUnit 5 a Testcontainers.
     *
```

```
     * @ServiceConnection (Spring Boot 3.1+) face un lucru remarcabil:
     * Spring Boot detecteaza automat containerul si configureaza
     * aplicatia sa se conecteze la el. Nu mai este necesara setarea
     * manuala a spring.datasource.url cu @DynamicPropertySource.
     *
     * Modificatorul static este important: indica un singur container
     * partajat intre toate testele clasei (eficienta).
     */
    @Container
    @ServiceConnection
    static MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0");

    @Autowired private TestRestTemplate restTemplate;
    @Autowired private ProductRepository productRepository;

    @BeforeEach
    void cleanDatabase() {
        productRepository.deleteAll();
    }

    @Test
    @DisplayName("creare si citire produs intr-o baza MySQL reala")
    void shouldCreateAndRetrieveProduct() {
        CreateProductRequest req = CreateProductRequest.builder()
                .name("Produs Testcontainers")
                .price(499.99).stock(20).build();

        ResponseEntity<ProductResponse> created = restTemplate.postForEntity(
                "/api/products", req, ProductResponse.class);

        assertThat(created.getStatusCode()).isEqualTo(HttpStatus.CREATED);
        assertThat(created.getBody().getId()).isNotNull();

        // Validam ca persistenta s-a propagat efectiv in MySQL,
        // nu doar in cache-ul Hibernate.
        assertThat(productRepository.count()).isEqualTo(1);

        ResponseEntity<ProductResponse> read = restTemplate.getForEntity(
                "/api/products/" + created.getBody().getId(),
                ProductResponse.class);

        assertThat(read.getStatusCode()).isEqualTo(HttpStatus.OK);
        assertThat(read.getBody().getName())
                  .isEqualTo("Produs Testcontainers");
    }
}
```

Câteva chei de înțelegere a acestui test:

- `@Testcontainers` (din pachetul `org.testcontainers.junit.jupiter`) este o extensie JUnit 5 care gestionează ciclul de viață al containerelor declarate cu `@Container`. Le pornește înainte de teste, le oprește la final, gestionează partajarea

(un container static este partajat între teste, unul instance-level este recreat pentru fiecare test).

- `MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0")` declară un container MySQL versiunea 8.0. La prima rulare pe o mașină nouă, imaginea Docker este descărcată (poate dura 1-2 minute); la următoarele rulări, este folosită din cache local și pornirea durează 5-10 secunde.
- `@ServiceConnection` – abstracția cheie a Spring Boot 3.1+. Fără ea, am fi avut nevoie de o metodă cu `@DynamicPropertySource` pentru a injecta `spring.datasource.url`, `.username`, `.password` în Environment-ul Spring; cu `@ServiceConnection`, totul se face automat. Codul este, în plus, generic: același pattern funcționează pentru PostgreSQL, MongoDB, Redis, Kafka etc.

Pentru a rula acest test, sunt necesare:

1. Docker instalat și pornit pe mașina de dezvoltare (Docker Desktop pe Windows/macOS, docker-engine pe Linux);
2. Acces la Docker Hub (sau un registry intern care oglindește imaginile MySQL);
3. Comanda `mvn verify` în terminal – Failsafe va detecta clasa `*IT` și o va rula.

Pe mașini cu Docker disponibil, testul rulează corect și verifică, suplimentar față de testul cu H2: comportamentul cu un dialect SQL real, interacțiunea cu un driver MySQL real (cu particularitățile lui privind tipurile, encoding-ul, fusurile orare etc.) și pornirea cu o schemă creată într-o bază reală.

11. Code coverage cu JaCoCo

După ce am scris suita de teste, putem folosi JaCoCo pentru a vedea cât din codul aplicației este efectiv exercitat. Plugin-ul a fost configurat în secțiunea de pom.xml; rămâne să rulăm comanda și să interpretăm raportul.

Comanda Maven care produce raportul este:

```
mvn clean verify
```

Aceasta rulează, în ordine: compilarea (`compile`), testele unitare (`test` – Surefire), testele de integrare (`integration-test` – Failsafe) și raportul JaCoCo (`verify` – execuția numită `report`). La final, raportul HTML este disponibil în:

```
target/site/jacoco/index.html
```

Deschis în browser, raportul oferă o vedere ierarhică: la nivel de proiect, sumarul total; la nivel de pachet, sumarele individuale; la nivel de clasă, lista metodelor; la nivel de metodă, vizualizarea liniilor de cod cu marcaje colorate (verde – linie acoperită, galben – branch parțial acoperit – roșu – linie neatinsă).

11.1. Cum interpretăm coverage-ul

Numărul total de coverage (de exemplu, „Total: 85%”) este informativ, dar nu este suficient. Mult mai utilă este analiza granulară: care clase au coverage scăzut? Sunt acelea critice pentru aplicație? Coverage-ul pe ramuri este, de regulă, mai relevant decât coverage-ul pe linii – o linie poate fi atinsă dar fără ca toate ramurile ei să fie testate.

Câteva reguli practice de interpretare:

- DTO-urile, modelele JPA și clasele Lombok-generate au, tipic, coverage natural foarte mare prin teste de integrare; nu este necesar să le testăm individual;
- logica de business (servicii, mappere, exception handlers) trebuie să aibă coverage ridicat (90%+), pentru că acolo apar bug-urile;
- clasele de configurare și `@Configuration` au, de regulă, coverage redus, dar acestea nu conțin logică testabilă în sens uzual;
- metodele simple (getteri/setteri non-Lombok, `equals/hashCode`) nu trebuie să primească teste dedicate doar pentru coverage.

Reamintim că coverage 100% nu garantează absența bug-urilor. Un test poate să atingă o linie fără a verifica nimic util. Coverage-ul ridicat înseamnă că, dacă există bug-uri în acele linii, există șansa ca testele să le prindă – dar numai dacă aserțiunile sunt suficient de stricte. Pentru o evaluare reală a calității testelor, instrumentul potrivit este mutation testing (PIT), nu doar coverage-ul.

11.2. Configurarea de praguri minime

Pentru proiecte mature, este util să configurăm JaCoCo astfel încât build-ul să eșueze dacă coverage-ul scade sub un prag stabilit. Aceasta previne degradarea silentă a calității testelor pe parcurs. Configurarea se face în `pom.xml`, în execuția `check` a plugin-ului:

```
<execution>
    <id>jacoco-check</id>
    <phase>verify</phase>
    <goals>
        <goal>check</goal>
    </goals>
    <configuration>
        <rules>
            <rule>
                <element>BUNDLE</element>
                <limits>
                    <limit>
                        <counter>LINE</counter>
                        <value>COVEREDRATIO</value>
                        <minimum>0.70</minimum>
                    </limit>
                </limits>
            </rule>
        </rules>
    </configuration>
</execution>
```

Dacă coverage-ul total pe linii scade sub 70%, build-ul eșuează. Pragul ales depinde de natura proiectului: 60-70% pentru aplicații obișnuite, 80%+ pentru produse cu cerințe ridicate de fiabilitate, 100% pentru sisteme critice (medical, aerospațial, financiar).

12. Bune practici și anti-patterns

Pentru a încheia partea teoretică, sintetizăm câteva reguli care s-au cristalizat în practica profesională a testării. Regulile sunt simple, dar respectarea lor consecventă face diferența dintre o suită de teste valoroasă și una care devine, treptat, o povară. [25]

12.1. Bune practici

12.1.1. Un test verifică un singur comportament

Un test bun se concentrează pe **un singur scenariu**. Numele lui descrie precis acel scenariu, iar în corp se găsesc, în general: o singură fază Act (un singur apel către SUT), una sau mai multe aserțiuni care verifică efectele acelui apel. Atunci când un test verifică prea multe lucruri, devine greu de înțeles când cade: care dintre cele cinci aserțiuni a eșuat? De ce? [25]

Antiteza este testul *epic*: un singur `@Test` care creează cinci produse, le actualizează, le caută, le șterge, verifică o duzină de stări intermediare. Dacă o aserțiune din mijloc cade, restul nu mai rulează, iar depanarea devine confuză. Soluția este descompunerea în mai multe teste mici, fiecare cu un scenariu clar.

12.1.2. Numirea expresivă a testelor

Numele unui test ar trebui să răspundă, singur, la întrebarea: **ce verifică?** Convenția cea mai populară este `metoda_cazul_rezultatul`: de exemplu, `findById_whenProductDoesNotExist_shouldThrowProductNotFoundException`. Acest format este lung dar precis. Combinat cu `@DisplayName` în limbaj natural, raportul de execuție devine inteligibil pentru oricine.

12.1.3. Independența totală a testelor

Niciun test nu trebuie să depindă de execuția altui test. JUnit 5 poate rula testele în orice ordine (chiar și în paralel), iar ordinea nu trebuie să afecteze rezultatul. Concret: nu folosim variabile statice partajate între teste (decât pentru constante imutabile), nu lăsăm date persistate de un test pentru altul, nu presupunem că un fișier creat în testul A va fi prezent în testul B. Adnotarea `@TestMethodOrder` există în JUnit 5, dar folosirea ei este, în general, un semn că testele nu sunt suficient de independente.

12.1.4. Fixture-uri minime și explicite

Fixture-urile (datele pregătite înainte de Act) ar trebui să fie **minime** – doar atât cât este necesar pentru testul respectiv. Un fixture umflat (un produs cu zece câmpuri, dintre care testul folosește doar două) face testul greu de citit și fragil. Atunci când mai multe teste partajează un fixture, putem extrage construcția într-o metodă de helper sau într-o clasă Object Mother (o clasă utilitară care creează obiecte de test), dar fără a centraliza prea mult:

dacă un test are nevoie de o variantă diferită, nu trebuie să se forțeze să folosească materul comun.

12.1.5. Aserțiuni expresive

Folosim AssertJ peste JUnit clasic ori de câte ori este posibil. `assertThat(list).contains("X")` este mai expresiv decât `assertTrue(list.contains("X"))`, iar mesajul de eroare în caz de eșec este mult mai util. În plus, aserțiunile înlănțuite (`assertThat(p).isNotNull().isEqualTo(expected)`) permit verificări multiple într-o singură expresie.

12.1.6. Don't mock what you don't own

Regula consacrată „nu mock-ui ce nu-ți aparține” spune că nu este bine să creezi mock-uri pentru clasele unor biblioteci externe (de exemplu, `RestTemplate`, `JdbcTemplate`, `EntityManager` direct). Motivul este că, dacă acea bibliotecă își schimbă comportamentul (chiar și subtil), mock-ul nu va reflecta schimbarea, dar codul de producție va fi afectat. Mai bine este să încapsulăm dependența externă într-un wrapper propriu (un service intern, un port în terminologia hexagonală) și să mock-uim wrapper-ul. În cazul testelor de integrare, folosim alternativa și mai bună: o instanță reală a dependenței prin Testcontainers sau echivalent.

12.1.7. Investește în mesaje de eroare bune

Atunci când un test cade, primul mesaj pe care îl citește dezvoltatorul este al aserțiunii. Un mesaj precis economisește minute de depanare. AssertJ produce mesaje foarte bune implicit, dar putem îmbunătăți și mai mult cu metoda `as()`:

```
assertThat(savedProduct.getId())
    .as("Id-ul ar trebui generat de baza dupa save()")
    .isNotNull();
```

12.2. Anti-patterns frecvente

12.2.1. Mocks care întorc mocks

Atunci când un mock returnează alt mock, care la rândul lui returnează alt mock, testul devine ilizibil și fragil. Acest pattern este, de regulă, un semn că structura codului SUT-ului este prea cuplată: SUT-ul cunoaște prea multe detalii ale colaboratorilor săi (*law of Demeter* încălcată). Soluția nu este o configurare mai atentă a mock-urilor, ci refactorizarea codului astfel încât SUT-ul să interacționeze doar cu colaboratorii săi imediați. [14]

12.2.2. Teste care testează implementarea, nu comportamentul

Un test care folosește excesiv `verify` ajunge să oglindească pas cu pas implementarea. Orice refactorizare internă, chiar și fără schimbare de comportament observabil, sparge testul. Acesta este un anti-pattern toxic: în loc să protejeze codul, testele inhibă schimbarea. Regula de aur este: **verifică intrarea-ieșirea, nu drumul intern**. `verify` este

indispensabil acolo unde nu există efect observabil decât prin interacțiune (apel către un serviciu extern, scriere în baza de date), dar trebuie folosit cu reținere.

12.2.3. Aserțiuni slabe sau lipsă

Un test care apelează SUT-ul dar nu verifică nimic, sau verifică doar că nu a aruncat o excepție, oferă protecție iluzorie. Coverage-ul va indica linia ca acoperită, dar bug-uri reale (rezultate incorecte) vor trece neprinse. În laboratorul de față, fiecare test face cel puțin câteva aserțiuni concrete asupra rezultatului sau asupra interacțiunilor.

12.2.4. Date magice de test

Numerele și șirurile aleator alese fără semnificație („`Product.builder().name("asdf").price(123.45).build()`”) fac testul greu de citit și de înțeles. Un test mai bun folosește valori care reflectă scenariul (`name("Tastatura mecanica").price(349.99)`), iar dacă e nevoie de variație, extrage constantele cu nume relevant: `private static final double DEFAULT_PRICE = 99.99;`.

12.2.5. Sleep-uri în teste

`Thread.sleep(500)` în mijlocul unui test este aproape întotdeauna un semn de probleme. Dacă testul are nevoie să aștepte un eveniment asincron, soluția este așteptarea deterministă (de exemplu, `Awaitility.await().atMost(5, SECONDS).until(condition)`), nu un sleep arbitrar. Un sleep prea scurt face testul flaky; unul prea lung încetinește suita. Folosirea Awaitility (sau echivalent) este preferabilă.

12.2.6. Ignorarea testelor (@Disabled)

`@Disabled` este util pentru a sări temporar peste un test cu probleme cunoscute, dar lăsarea lui pe termen lung este toxică: testul nu mai oferă protecție, iar timpul investit în el a fost irosit. Regula este: când un test este `@Disabled`, se deschide imediat un ticket pentru a-l rezolva, iar el nu trece niciodată într-o ramură de release.

12.2.7. Teste foarte lente

O suită de teste care durează 10 minute va fi rulată rar de programatori, ceea ce reduce drastic valoarea ei. Cauzele frecvente: prea multe teste de integrare în loc de teste unitare, fixture-uri complexe inițializate la fiecare test (în loc de partajate cu `@BeforeAll` când e sigur), porniri repetate ale contextului Spring (Spring Boot face caching între teste, dar diferențe minore între adnotări forțează recrearea). Optimizarea timpului de execuție al suitei este o investiție continuă.

13. Întrebări recapitulative

13.1. Concepte fundamentale

1. Care este diferența esențială dintre un test unitar și un test de integrare? Care sunt avantajele și dezavantajele fiecărei categorii?

2. Ce reprezintă piramida testării? Ce tipuri de teste se află la baza ei și care în vârf? Justificați raportul propus.
3. Care este anti-patternul „ice cream cone” și de ce este considerat dăunător?
4. Ce este SUT (System Under Test)? Cum identificați SUT-ul într-un test dat?
5. Explicați șablonul Arrange-Act-Assert. De ce este recomandată respectarea lui în fiecare test?
6. Ce înseamnă „izolarea testelor”? Ce probleme apar atunci când izolarea este încălcată?
7. Ce sunt flaky tests? Care sunt cele mai frecvente cauze și cum se evită?
8. Ce este code coverage? De ce un coverage de 100% nu garantează absența bug-urilor?

13.2. Test doubles

1. Enumerați cele cinci tipuri de test doubles din taxonomia Meszaros. Care este caracteristica esențială a fiecăruia?
2. Care este diferența dintre un mock și un stub? Dați un exemplu pentru fiecare folosind clasele din proiect.
3. Când este preferabilă verificarea de stare (state-based) și când verificarea comportamentală (interaction-based)?
4. De ce am folosit, în testele ProductService, un SimpleMeterRegistry real în loc de un mock al MeterRegistry?
5. Ce este un ArgumentCaptor și în ce situație este util? Dați un exemplu din testele scrise.
6. Ce face metoda verifyNoInteractions() și de ce este utilă?
7. Care este diferența dintre @Mock și @Spy în Mockito?

13.3. Spring Boot Test slices

1. Ce diferențe esențiale există între @SpringBootTest, @WebMvcTest și @DataJpaTest? Pentru fiecare, ce componente Spring sunt încărcate?
2. Ce face @MockBean? Cum diferă de @Mock obișnuit?
3. De ce este necesar @TestConfiguration cu un MeterRegistry în testele @WebMvcTest pentru ProductController?
4. Ce este TestEntityManager? Ce avantaj oferă față de a folosi direct repository-ul în pregătirea fixture-urilor?
5. Care este diferența dintre MockMvc și TestRestTemplate? Când este recomandat fiecare?
6. Ce este JSONPath? Dați exemple de expresii folosite în testele MockMvc.

13.4. Configurare și execuție

1. Care este rolul plugin-urilor maven-surefire-plugin și maven-failsafe-plugin? Ce convenții de denumire impune fiecare?

2. De ce am separat testele de integrare cu Testcontainers (cu sufixul *IT) de testele unitare (cu sufixul *Test)?
3. Ce conține fișierul src/test/resources/application.properties? De ce este preferată o bază H2 in-memory pentru teste?
4. Ce face plugin-ul JaCoCo? Cum se rulează și unde se găsește raportul?
5. Cum se configurează JaCoCo să eșueze build-ul dacă coverage-ul scade sub un prag minim?

13.5. Testcontainers

1. Ce probleme rezolvă Testcontainers față de testarea cu H2 in-memory?
2. Ce face adnotarea `@ServiceConnection` (Spring Boot 3.1+)?
3. De ce este declarată variabila `MySQLContainer` ca static în testul `ProductTestcontainersIT`? Ce s-ar întâmpla fără modificator?
4. Ce cerințe de mediu (instalări, acces) sunt necesare pentru rularea testelor Testcontainers?

13.6. Bune practici

1. Care sunt cele mai importante reguli pentru numirea testelor? Ilustrați cu exemple din proiect.
2. De ce este recomandată folosirea AssertJ în locul aserțiunilor JUnit clasice? Dați exemple comparative.
3. Ce înseamnă regula „don't mock what you don't own”? Cum se respectă în practică?
4. Identificați cel puțin trei anti-patterns frecvente în testare și explicați de ce fiecare este problematic.
5. Ce este un „smoke test” și ce valoare oferă, deși pare gol?
6. Ce sunt test smells? Dați câteva exemple concrete.

14. Exerciții propuse

14.1. Exercițiul 1 – Teste parametrizate pentru validare

Testele de validare existente (`createProduct_whenNameIsBlank`, `createProduct_whenPriceIsZero` etc.) sunt foarte similare: aceeași structură, doar valorile diferă. Refactorizați-le într-un singur `@ParameterizedTest` care primește, dintr-un `@MethodSource` sau `@CsvSource`, combinații de valori invalide și câmpul așteptat în răspuns. Verificați, pentru fiecare, că răspunsul este 400 și că eroarea apare pe câmpul corect.

Schelet de pornire:

```
@ParameterizedTest(name = "validare invalida: {0}")
@MethodSource("invalidRequestsProvider")
@DisplayName("POST /api/products - validari multiple")
void createProduct_whenInvalid_shouldReturn400(
        String description, CreateProductRequest request,
```

```
            String expectedErrorField) throws Exception {
        // ...
    }

    private static Stream<Arguments> invalidRequestsProvider() {
        return Stream.of(
            Arguments.of("nume gol",

    CreateProductRequest.builder().name("").price(50.0).stock(10).build(),
                "name"),
            Arguments.of("pret zero",


    CreateProductRequest.builder().name("X").price(0.0).stock(10).build(),
                "price"),
            // ... completati restul cazurilor
        );
    }
```

14.2. Exercițiul 2 – Test pentru endpoint-ul Actuator de metrici

Aplicația expune endpoint-uri Actuator (`/actuator/metrics`, `/actuator/health`). Scrieți un test de integrare `@SpringBootTest` care: creează un produs prin POST, apoi face GET pe `/actuator/metrics/products.created.count` și verifică în răspunsul JSON că măsurarea `COUNT` este 1. Repetați pentru două creări și verificați că este 2.

14.3. Exercițiul 3 – Migrare la @MockitoBean (Spring Boot 3.4+)

Spring Boot 3.4 a introdus adnotările `@MockitoBean` și `@MockitoSpyBean` pentru a înlocui `@MockBean` (deprecated). Migrarea este simplă, dar trebuie făcută conștient. Actualizați versiunea Spring Boot a proiectului la 3.4+ în `pom.xml`, înlocuiți toate aparițiile `@MockBean` cu `@MockitoBean` și verificați că suita continuă să treacă. Notați diferențele de comportament observate (de regulă nu există, dar pot apărea probleme la cache-ul de context Spring).

14.4. Exercițiul 4 – Tagging și execuție selectivă a testelor

JUnit 5 oferă adnotarea `@Tag` pentru a clasifica testele în categorii. Adăugați în proiect tag-uri precum `"fast"`, `"slow"` și `"integration"`, marcați fiecare test corespunzător și configurați două profile Maven distincte:

- profile-ul fast – rulează doar testele cu `@Tag("fast")`;
- profile-ul all – rulează tot.

Configurarea se face în `pom.xml` prin `<groups>` în Surefire (`includeTags` / `excludeTags`). Verificați rularea cu `mvn -P fast test`.

Laborator 7: Introducere în React. Componente, JSX, props și state

1. Obiectivul laboratorului

În acest laborator se urmărește înțelegerea modelului de programare **React**: o bibliotecă JavaScript pentru construirea de interfețe utilizator pe baza conceptului de componentă. Punctul de plecare este observația că, până la acest moment, frontend-ul aplicației REST construite în laboratoarele anterioare a fost limitat la mijloace improvizate – afișarea brută a JSON-ului în browser, sau folosirea unor clienți de tip Postman. Pentru o aplicație utilizabilă în practică, este nevoie de o interfață grafică, iar React este una dintre cele mai utilizate biblioteci pentru construirea unor astfel de interfețe. [7, 30]

Laboratorul tratează în mecanismele fundamentale ale React: definirea unei componente, sintaxa JSX, transmiterea datelor prin props, gestionarea stării locale prin `useState`, tratarea evenimentelor și compoziția componentelor într-un arbore.

La finalul laboratorului, cititorul ar trebui să fie capabil:

- să explice conceptul de aplicație SPA (Single Page Application) și să motiveze de ce un API REST se potrivește cu acest model;
- să instaleze instrumentele necesare dezvoltării frontend cu React: Node.js, npm, Visual Studio Code și extensiile uzuale;
- să creeze un proiect React folosind Vite și să înțeleagă structura de directoare generată;
- să scrie o componentă funcțională simplă, folosind sintaxa JSX;
- să transmită date către o componentă prin props și să le folosească în interiorul ei;
- să declare și să modifice starea locală a unei componente folosind hook-ul useState;
- să trateze evenimentele uzuale (click, schimbare în input) și să înțeleagă mecanismul de re-randare;
- să compună componente într-un arbore, transmițând date de la părinte la copii și acțiuni de la copii spre părinte.

2. De ce React

Aplicația REST construită în laboratoarele anterioare returnează date în format JSON. Aceste date trebuie afișate undeva, într-o formă lizibilă pentru utilizator: un tabel cu produse, un formular de adăugare, un buton de ștergere.

Există, în mare, două abordări tradiționale pentru a obține o interfață grafică care interacționează cu un backend.

Prima abordare este **randarea pe server** (server-side rendering). Serverul generează HTML-ul direct, ca în paginile clasice JSP[24], Thymeleaf[25] sau PHP[26]. Avantajul este simplitatea: tot codul este într-un singur loc, iar livrarea către client este o pagină completă, gata de afișat. Dezavantajul major este că fiecare interacțiune (click pe un buton, sortare a unui tabel, deschidere a unui formular) presupune un ciclu cerere-răspuns complet către server și o re-randare integrală a paginii. Experiența utilizatorului este fragmentată (încărcarea completă a unei noi pagini presupune, de obicei, trecere print-o pagină albă), iar serverul este încărcat cu generarea HTML-ului pentru fiecare interacțiune.

A doua abordare este **aplicația de tip SPA** (Single Page Application). Serverul livrează o singură pagină HTML, însoțită de codul JavaScript al aplicației; toată logica de afișare se desfășoară în browser, iar comunicarea cu serverul se reduce la apeluri JSON pe API-ul REST. Navigarea între „pagini" nu mai presupune cereri către server pentru pagini noi, ci doar schimbarea componentelor afișate, ceea ce este mult mai rapid. Avantajul este o experiență fluidă, similară unei aplicații desktop, și o separare curată între backend (date) și frontend (prezentare). Dezavantajul este complexitatea suplimentară din partea de client.

Atunci când backend-ul este deja un API REST, modelul SPA este alegerea naturală: serverul nu se mai ocupă deloc de prezentare, iar frontend-ul poate evolua independent, fiind dezvoltat eventual de o echipă diferită.

React a fost creat de Facebook în 2013 și a devenit, în timp, una dintre cele mai utilizate biblioteci pentru construirea de SPA-uri. Două idei fundamentale stau la baza ei. Prima este *modelul de componente*: interfața este descrisă ca un arbore de componente, fiecare componentă fiind o funcție care, pe baza unor date de intrare, produce o descriere a unei bucăți de UI. A doua este *DOM-ul virtual*: React nu modifică direct nodurile reale ale paginii la fiecare schimbare, ci păstrează în memorie o reprezentare zveltă (virtual DOM[27]), o compară cu cea anterioară și aplică doar diferențele necesare. Această strategie reduce numărul de operații DOM efective și face aplicația mai rapidă.

2.1. React în peisajul actual

React nu este singura opțiune. Concurenți direcți, cu modele similare, sunt Vue.js (folosit pe scară largă, cu o curbă de învățare mai blândă), Angular (framework complet de la Google), Svelte (compilator care generează cod JavaScript optim, fără DOM virtual).

[24] JSP este o tehnologie Java pentru generarea paginilor web dinamice pe server, prin combinarea codului Java cu HTML.

[25] Thymeleaf este un motor de template-uri folosit frecvent cu Spring Boot, care permite generarea de pagini HTML dinamice prin inserarea datelor din aplicație în șabloane HTML.

[26] PHP este un limbaj de programare server-side folosit pentru dezvoltarea aplicațiilor web dinamice, unde codul PHP generează conținut HTML trimis către browser.

[27] DOM (Document Object Model) este o reprezentare în memorie, sub formă de arbore de obiecte, a unui document HTML sau XML, care permite manipularea structurii, conținutului și stilului acestuia prin cod (de exemplu JavaScript).

3. Instrumente necesare și instalarea lor

Pentru a porni dezvoltarea unei aplicații React sunt necesare câteva instrumente, prezentate mai jos. Toate sunt disponibile gratuit pe Windows, macOS și Linux. Ordinea de instalare recomandată este aceea în care apar mai jos.

3.1. Node.js și npm

Node.js este un mediu de execuție JavaScript construit deasupra motorului V8 al browserului Chrome, dar care rulează în afara browserului. Pentru frontend, Node.js nu este folosit la rularea efectivă a aplicației React (aceasta rulează în browser), ci pentru *lanțul de instrumente de dezvoltare*: serverul de dezvoltare cu hot reload, transformarea JSX-ului în JavaScript valid, optimizarea bundle-ului final, instalarea bibliotecilor terțe. Toate aceste utilitare sunt scrise în JavaScript și au nevoie de Node pentru a rula. [12]

npm (Node Package Manager) este managerul de pachete oficial al ecosistemului Node.js. Se instalează automat odată cu Node.js și permite instalarea de biblioteci dintr-un registru public[28] printr-o singură comandă (de exemplu `npm install react`). Există și alternative populare (yarn, pnpm) care sunt funcțional echivalente.

3.1.1. Instalare pe Windows

Se accesează site-ul oficial nodejs.org și se descarcă instalatorul .msi pentru versiunea LTS (Long Term Support) curentă (la data redactării acestui îndrumar, versiunea 20.x sau 22.x). Versiunea LTS este recomandată pentru proiecte serioase pentru că primește actualizări de securitate timp de aproximativ 30 de luni de la lansare; versiunea „Current” conține cele mai noi funcționalități, dar nu este garantat stabilă pentru o perioadă lungă.

La rularea instalatorului, se acceptă valorile implicite. Casetele „Add to PATH” și „Install npm package manager” trebuie să rămână bifate (sunt bifate implicit). Caseta opțională „Tools for Native Modules” (care instalează Python și compilatoare C++) poate fi bifată dacă există spațiu pe disc; nu este strict necesară pentru lucrul cu React, dar poate evita ulterior probleme la instalarea unor pachete care includ cod nativ.

3.1.2. Instalare pe macOS

Cea mai simplă cale este prin managerul de pachete **Homebrew**. Dacă acesta nu este deja instalat, se rulează scriptul oficial din terminal:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

După instalarea Homebrew, se rulează:

```
brew install node
```

[28] Registrul public npm este disponibil la adresa https://www.npmjs.com și conține peste două milioane de pachete.

Alternativ, ca pe Windows, se poate descărca instalatorul .pkg de pe nodejs.org. Avantajul Homebrew este că permite ușor schimbarea versiunii (`brew upgrade node`) și instalarea altor utilitare uzuale.

3.1.3. Instalare pe Linux (Ubuntu / Debian)

Pe distribuțiile bazate pe Debian, versiunea din depozitele oficiale ale sistemului este, de regulă, prea veche. Se recomandă folosirea depozitului oficial NodeSource:

```
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt-get install -y nodejs
```

Pentru alte distribuții (Fedora, Arch, openSUSE), instrucțiunile concrete sunt disponibile pe pagina `https://nodejs.org/en/download/package-manager`.

3.1.4. Verificarea instalării

Indiferent de sistemul de operare, după instalare se deschide un terminal nou (terminalele deja deschise nu „văd” modificările PATH) și se rulează:

```
node -v
npm -v
```

Prima comandă afișează versiunea Node.js instalată (de exemplu v20.11.1), a doua versiunea npm (de exemplu 10.2.4). Dacă apare o eroare de tip „command not found”, instalarea nu este completă sau PATH-ul nu este configurat corect; cea mai simplă soluție este reinstalarea, asigurând opțiunea „Add to PATH”.

> **Notă privind versiunile**
>
> Pe parcursul unui proiect, versiunea Node.js trebuie să rămână relativ constantă (toți membrii echipei și serverul de CI ar trebui să folosească aceeași versiune majoră). Pentru proiecte care necesită mai multe versiuni Node simultan (de exemplu, un proiect vechi pe Node 16 și unul nou pe Node 20) există utilitare specializate: `nvm` pe Linux/macOS și `nvm-windows` pe Windows. În cadrul acestui curs nu este nevoie de ele.

3.2. Visual Studio Code și extensiile recomandate

Visual Studio Code (cunoscut și ca „VS Code”) este un editor de text gratuit, open-source, dezvoltat de Microsoft. Este în prezent cel mai folosit editor în industrie pentru dezvoltare web, datorită combinației dintre performanță, suport excelent pentru JavaScript/TypeScript și un ecosistem foarte mare de extensii.

Se descarcă de la `https://code.visualstudio.com` și se instalează cu opțiunile implicite. Pe Windows, este recomandat să se bifeze, în timpul instalării, „Add ‘Open with Code’ action to Windows Explorer file/directory context menu” – permite deschiderea rapidă a unui director ca proiect, prin click dreapta.

3.2.1. Extensii recomandate

Extensiile se instalează din panoul Extensions (Ctrl+Shift+X) căutând după nume. Pentru lucrul cu React sunt utile, în ordinea importanței:

- **ESLint** (publisher: Microsoft) – integrează în editor verificatorul ESLint, care semnalează în timp real probleme de stil și greșeli uzuale (variabile nefolosite, dependențe lipsă în `useEffect`, reguli React încălcate). Vite generează un proiect cu ESLint deja configurat; extensia doar afișează rezultatele direct în cod.
- **Prettier - Code formatter** (publisher: Prettier) – formatator automat de cod. Se configurează să ruleze la salvare (File → Preferences → Settings → căutare „Format On Save", bifat). Avantajul major este că toți membrii echipei produc cod formatat identic, indiferent de stilul personal al fiecăruia.
- **ES7+ React/Redux/React-Native snippets** (publisher: dsznajder) – oferă șabloane scurte (snippets) pentru construcții uzuale: `rfc` urmat de Tab generează scheletul unei componente funcționale, `useState` generează o declarație de stare. Reduce sensibil scrisul mecanic.
- **Auto Rename Tag** (publisher: Jun Han) – când redenumiți tag-ul de deschidere, îl redenumește automat și pe cel de închidere. Util în JSX, unde tag-urile compuse apar des.
- **Material Icon Theme** sau **vscode-icons** – temă de iconițe pentru fișiere și directoare. Strict cosmetică, dar ajută la orientare în proiectele cu multe fișiere.

3.2.2. Setări utile

Câteva setări care merită activate, prin File → Preferences → Settings (sau, direct în `settings.json`, cu `Ctrl+Shift+P` → „Preferences: Open User Settings (JSON)"):

```
{
  "editor.formatOnSave": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "editor.tabSize": 2,
  "files.eol": "\n",
  "javascript.updateImportsOnFileMove.enabled": "always"
}
```

Tab-size 2 este convenția uzuală în JavaScript și React (spre deosebire de Java, unde se folosesc 4 spații). Convertirea automată a importurilor la mutarea unui fișier evită erorile manuale când se restructurează proiectul.

3.3. Browser și React Developer Tools

Pentru rularea aplicației este suficient orice browser modern (Chrome, Firefox, Edge, Safari). Recomandarea este Chrome sau Firefox, pentru extensia React Developer Tools, care permite inspectarea arborelui de componente, a props-urilor și a stării – similar cu modul în care DevTools clasic permite inspectarea DOM-ului.

Pentru Chrome / Edge: *Chrome Web Store → căutare „React Developer Tools"* → Add to Chrome. Pentru Firefox: *addons.mozilla.org → căutare „React Developer Tools"*. După instalare, în panoul DevTools al browserului apar două taburi noi: `Components` (arborele componentelor) și `Profiler` (analiza performanței re-randărilor).

3.4. Instrumente opționale

Git – sistem de versionare a codului. Nu este strict necesar pentru a rula exemplele din acest îndrumar, dar este indispensabil pentru orice proiect serios. Se descarcă de la `https://git-scm.com`. Configurarea minimă, după instalare:

```
git config --global user.name "Numele dvs"
git config --global user.email "email@example.com"
```

Postman sau **Bruno** – clienți REST grafici, deja folosiți în laboratoarele anterioare pentru testarea manuală a backend-ului. Rămân utili în acest laborator pentru a verifica rapid că serverul Spring Boot răspunde corect, înainte de a încerca apeluri din React.

4. Pornirea proiectului cu Vite

Vite (pronunțat /viːt/, de la cuvântul francez pentru „rapid") este un instrument modern pentru crearea, dezvoltarea și construirea aplicațiilor web în JavaScript. A fost creat de Evan You – autorul Vue.js – și a devenit, după 2022, alegerea standard pentru proiectele React noi, înlocuind Create React App[29] care este în prezent considerat depășit. [8, 43, 44]

Vite oferă două lucruri esențiale: un server de dezvoltare cu pornire instantanee și *hot module replacement* (HMR), și un sistem de build optimizat (bazat pe Rollup) care produce fișierele statice livrabile în producție. Aceste două roluri sunt decuplate intenționat – ce este rapid în dezvoltare nu e neapărat optim pentru livrare, și invers.

4.1.1. Crearea proiectului

Crearea unui proiect React nou se face printr-o singură comandă, dintr-un terminal poziționat în directorul în care dorim să creăm proiectul (de exemplu, `C:\Users\dvs\proiecte` pe Windows sau `~/projects` pe Linux/macOS):

```
npm create vite@latest products-ui -- --template react
```

Argumentele sunt: `products-ui` – numele directorului în care va fi creat proiectul (poate fi orice nume valid de director); `--template react` – Vite suportă mai multe template-uri (React, Vue, Svelte, vanilla JavaScript etc.); cel folosit aici este React cu JavaScript pur.

Notă: cele două `--` separate (din `create vite@latest products-ui -- --template`) sunt obligatorii. Primul `--` spune lui npm „opțiunile care urmează sunt pentru programul rulat, nu pentru npm însuși".

La prima rulare, dacă utilitarul Vite nu este disponibil local, npm îl descarcă temporar din registru (mecanismul npx). Procesul durează 10–20 de secunde, apoi se afișează:

[29] Create React App (CRA) a fost utilitarul oficial pentru crearea proiectelor React între 2016 și 2023. A fost retras din mentenanță activă din cauza performanței lente și a configurației rigide. Vite este de aproximativ 10–20 de ori mai rapid la pornire și la hot reload.

```
Scaffolding project in ./products-ui...

Done. Now run:

  cd products-ui
  npm install
  npm run dev
```

Comanda `npm install` citește fișierul `package.json` (generat de Vite) și descarcă toate dependențele declarate acolo în directorul `node_modules/`.

4.1.2. Structura proiectului generat

După npm install, structura completă a directorului products-ui este:

```
products-ui/
├── node_modules/         ← bibliotecile descărcate (sute de directoare)
├── public/               ← fișiere statice servite ca atare
│   └── vite.svg
├── src/                  ← codul aplicației (aici lucrăm)
│   ├── assets/
│   │   └── react.svg
│   ├── App.css
│   ├── App.jsx           ← componenta principală
│   ├── index.css
│   └── main.jsx          ← punctul de intrare
├── .gitignore
├── eslint.config.js      ← configurație ESLint
├── index.html            ← pagina HTML servită de Vite
├── package.json          ← descrierea proiectului și dependențele
├── package-lock.json     ← versiunile exacte instalate
└── vite.config.js        ← configurația Vite
```

Câteva observații importante despre fișierele cheie:

4.1.3. index.html

Spre deosebire de un proiect tradițional, `index.html` nu se află într-un director `public/` ascuns, ci la rădăcina proiectului. Vite tratează această pagină ca punct de intrare al aplicației. Conținutul ei este simplu:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/vite.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Vite + React</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
</html>
```

Două elemente sunt esențiale: `<div id="root"></div>` este containerul în care React va injecta arborele de componente, iar `<script type="module" src="/src/main.jsx">` indică punctul de intrare al codului JavaScript.

4.1.4. src/main.jsx

Acesta este punctul de intrare al codului React. Conținutul implicit este:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import "./index.css";
import App from "./App.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <App />
  </StrictMode>,
);
```

Funcția `createRoot` preia elementul DOM cu id-ul `root` (cel din `index.html`) și îl marchează ca rădăcină a aplicației React. Apelul `.render(...)` îi spune lui React să randeze, în acest container, arborele de componente descris ca argument.

`<StrictMode>` este o componentă specială React care, în modul de dezvoltare, activează verificări suplimentare: identifică efecte secundare problematice, apeluri ale unor API-uri vechi, etc. În producție, `<StrictMode>` nu are niciun efect. Una dintre consecințele ei vizibile este că efectele `useEffect` sunt rulate de două ori în dezvoltare (pentru a depista efecte care nu se curăță corect); în producție, asta nu se întâmplă.

4.1.5. src/App.jsx

Componenta de la rădăcina aplicației. Conținutul generat de Vite este o pagină demo care va fi înlocuită complet pe parcursul laboratorului. Structura este o funcție care întoarce JSX:

```
import { useState } from "react";
import reactLogo from "./assets/react.svg";
import viteLogo from "/vite.svg";
import "./App.css";

function App() {
  const [count, setCount] = useState(0);
  return (
    <>
      <div>
        <a href="https://vite.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        {/* ... restul demoului ... */}
      </div>
      <h1>Vite + React</h1>
      <button onClick={() => setCount((c) => c + 1)}>
        count is {count}
      </button>
```

```
      </>
    );
  }

  export default App;
```

4.1.6. package.json

Acest fișier descrie proiectul și conține lista de dependențe și scripturile npm disponibile:

```
{
  "name": "products-ui",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev":     "vite",
    "build":   "vite build",
    "lint":    "eslint .",
    "preview": "vite preview"
  },
  "dependencies": {
    "react":     "^18.3.1",
    "react-dom": "^18.3.1"
  },
  "devDependencies": {
    "@vitejs/plugin-react": "^4.3.1",
    "eslint":               "^9.9.0",
    "vite":                 "^5.4.1"
  }
}
```

Cele patru scripturi sunt comenzile uzuale ale fluxului de lucru:

- **`npm run dev`** – pornește serverul de dezvoltare cu hot reload, ascultă implicit pe portul 5173. Folosit zilnic, în timpul dezvoltării.
- **`npm run build`** – produce build-ul de producție în directorul `dist/` (HTML, JS și CSS minificate). Se rulează atunci când dorim să livrăm aplicația.
- **`npm run preview`** – pornește un server local care servește build-ul de producție. Se folosește pentru a verifica, înainte de a livra, că build-ul se comportă corect.
- **`npm run lint`** – rulează ESLint asupra întregului proiect și raportează probleme. Util în CI, dar și manual înainte de un commit.

Distincția dintre `dependencies` și `devDependencies` este importantă: primele sunt biblioteci necesare la rularea aplicației finale (în browser), celelalte doar în timpul dezvoltării (compilator, linter, server de dev). Această separare ajută la generarea unui build de producție care nu include codul instrumentelor.

4.2. Pornirea serverului de dezvoltare

Cu proiectul instalat, se rulează:

```
cd products-ui
npm run dev
```

În terminal, se afișează ceva similar cu:

```
VITE v5.4.1  ready in 412 ms

➜  Local:   http://localhost:5173/
➜  Network: use --host to expose
➜  press h + enter to show help
```

Se deschide în browser `http://localhost:5173` și apare pagina demo cu logo-urile Vite și React, plus un buton care numără click-urile. Hot reload este deja activ: orice modificare în fișierele sursă se reflectă instantaneu în browser, fără refresh manual.

> **Atenție**
>
> Serverul de dezvoltare trebuie să ruleze permanent în timpul lucrului (terminalul respectiv rămâne ocupat). Pentru a-l opri, se apasă **`Ctrl+C`** în terminal. Pentru a executa alte comenzi (de exemplu **`npm install`** pentru o bibliotecă nouă), se deschide un al doilea terminal.

4.3. Curățarea proiectului demo

Înainte de a începe efectiv, este util să curățăm pagina demo, lăsând o componentă App minimă, ca punct de plecare. Modificăm src/App.jsx astfel:

```
import "./App.css";

function App() {
  return (
    <div>
      <h1>Catalog produse</h1>
    </div>
  );
}

export default App;
```

Ștergem din `src/App.css` și `src/index.css` conținutul demo (lăsându-le goale, le vom umple gradual). De asemenea, putem șterge fișierele `src/assets/react.svg` și `public/vite.svg`, precum și importurile lor – nu vor mai fi referite. După salvare, pagina din browser se actualizează automat și afișează doar titlul „Catalog produse”. Avem un punct de plecare curat.

5. Anatomia unei componente React

Componenta este unitatea fundamentală de organizare a unei aplicații React. În varianta modernă a bibliotecii (post 2019), o componentă este, pur și simplu, *o funcție JavaScript care returnează JSX*. Variantele mai vechi (componente declarate ca clase) sunt încă suportate, dar nu se mai folosesc în cod nou. [7, 30]

Cea mai simplă componentă posibilă arată astfel:

```
function Salut() {
  return <h1>Salut din React!</h1>;
}

export default Salut;
```

Pentru a folosi această componentă, o salvăm într-un fișier nou `src/Salut.jsx`, o importăm în `App.jsx` și o includem în JSX-ul componentei App:

```
import "./App.css";
import Salut from "./Salut.jsx";

function App() {
  return (
    <div>
      <h1>Catalog produse</h1>
      <Salut />
    </div>
  );
}

export default App;
```

5.1. Reguli de bază pentru componente

Numele unei componente începe obligatoriu cu literă mare. Aceasta nu este o convenție opțională, ci o regulă a JSX-ului: tag-urile care încep cu literă mică sunt interpretate ca elemente HTML obișnuite (`<div>`, `<button>`), iar cele care încep cu literă mare sunt interpretate ca referințe la o componentă React. `<salut />` ar fi interpretat ca element HTML „salut” (necunoscut), nu ca apelul componentei noastre. [7]

O componentă trebuie să returneze un singur element rădăcină. Codul

```
function Salut() {
  return (
    <h1>Salut!</h1>
    <p>Bun venit.</p>
  );  // EROARE
}
```

produce o eroare la compilare. Soluția este fie încadrarea în `<div>` sau alt element, fie folosirea unui *fragment* – sintaxa specială `<>...</>` care permite gruparea fără a introduce un nod DOM suplimentar:

```
function Salut() {
  return (
    <>
      <h1>Salut!</h1>
      <p>Bun venit.</p>
    </>
  );
}
```

O componentă este, în esență, o funcție pură. Pentru aceleași date de intrare (props, state), trebuie să producă același JSX. Apeluri către API-uri externe, modificări ale unor variabile globale, accesul la `document.getElementById` și operații similare nu trebuie să se afle în corpul direct al funcției – pentru ele există mecanisme dedicate (hook-uri, în special `useEffect`).

6. JSX în detaliu

JSX este o extensie de sintaxă pentru JavaScript, propusă de echipa React, care permite scrierea unor structuri arborescente de noduri într-o formă similară HTML-ului. JSX nu este HTML, și nu este nici JavaScript valid: el este transformat, în timpul compilării, în apeluri JavaScript obișnuite către funcția `React.createElement`. De exemplu, expresia

```
<h1 className="titlu">Salut!</h1>
```

este transformată, în spatele scenei, în:

```
React.createElement("h1", { className: "titlu" }, "Salut!")
```

Pentru programator, această transformare este invizibilă; el scrie JSX și uită că în spate există un apel de funcție. Dar înțelegerea echivalenței ajută la rezolvarea unor situații confuze: de exemplu, o componentă produce un „element React” (un obiect JavaScript), nu un nod DOM real. Diferența este ce permite DOM-ului virtual să funcționeze. [7, 15]

6.1. Diferențe față de HTML

Deși JSX seamănă cu HTML, are câteva diferențe sintactice obligatorii, justificate de faptul că, în spate, se transformă în JavaScript [7, 16]:

HTML	*JSX*	*Motivul*
`class="..."`	`className="..."`	`class` este cuvânt rezervat în JS
`for="..."`	`htmlFor="..."`	`for` este cuvânt rezervat în JS
`onclick`	`onClick`	Convenția JSX: camelCase pentru atribute
` `	` `	Toate tag-urile fără conținut trebuie auto-închise
`style="color:red"`	`style={{color:"red"}}`	style primește un obiect, nu un șir
`<!-- comentariu -->`	`{/* comentariu */}`	În JSX, comentariile sunt expresii JS

6.2. Expresii JavaScript în JSX

În interiorul JSX-ului, acoladele `{ ... }` marchează o expresie JavaScript a cărei valoare va fi inserată în arbore. Orice expresie JavaScript validă este permisă: variabile, calcule, apeluri de funcții, expresii ternare. [15, 27, 28]

```
function Salut() {
  const nume = "Ana";
  const ora = new Date().getHours();
  const este_zi = ora >= 6 && ora < 22;

  return (
    <div>
      <h1>Salut, {nume.toUpperCase()}!</h1>
      <p>Acum este ora {ora}.</p>
      <p>{este_zi ? "Bună ziua!" : "Bună seara!"}</p>
      <p>Suma: {2 + 3}</p>
    </div>
  );
}
```

Atenție: în interiorul `{ }` se așteaptă o *expresie* (ceva care produce o valoare), nu o *instrucțiune*. Astfel, `{ if (x) ... }` sau `{ for (let i...) ... }` nu sunt permise. Pentru condiții se folosește operatorul ternar (`?:`) sau operatorul `&&`, iar pentru iterații, metoda `Array.prototype.map()`.

6.3. Randare condițională

Există patru modalități uzuale de a randa condițional un element în React:

6.3.1. Operatorul ternar

```
<p>{user ? "Bun venit, " + user.nume : "Vă rugăm să vă autentificați."}</p>
```

6.3.2. Operatorul && (short-circuit)

```
{produs.stock === 0 && <span className="alert">Indisponibil</span>}
```

Atenție: dacă valoarea din stânga lui `&&` este `0` (un număr), JSX-ul va randa „0” în pagină. Acesta este o capcană clasică. Soluția este să se asigure că în stânga este o valoare booleană, de exemplu `produse.length > 0 && ...` în loc de `produse.length && ...`.

6.3.3. Întoarcere timpurie (early return)

Pentru cazurile mari (de exemplu, afișarea unui mesaj de încărcare în loc de conținut), se folosește o întoarcere timpurie din funcție:

```
function ProductsList({ products, loading }) {
  if (loading) return <p>Se încarcă...</p>;
  if (products.length === 0) return <p>Nu există produse.</p>;
  return (
    <ul>
      {products.map(p => <li key={p.id}>{p.name}</li>)}
```

```
      </ul>
    );
  }
```

6.3.4. Variabile JSX

JSX-ul poate fi atribuit unei variabile și apoi inserat:

```
let mesaj;
if (loading) {
  mesaj = <p>Se încarcă...</p>;
} else if (error) {
  mesaj = <p className="err">Eroare: {error.message}</p>;
} else {
  mesaj = <Lista produse={produse} />;
}
return <div>{mesaj}</div>;
```

6.4. Randarea unei liste cu .map()

Pentru a randa o listă de elemente, se folosește metoda `Array.prototype.map`. Pentru fiecare element din tablou se produce un fragment de JSX, iar întreaga colecție rezultată este inserată în arbore:

```
const produse = [
  { id: 1, name: "Tastatură" },
  { id: 2, name: "Monitor" },
  { id: 3, name: "Mouse" },
];

function Lista() {
  return (
    <ul>
      {produse.map(p => <li key={p.id}>{p.name}</li>)}
    </ul>
  );
}
```

Atributul `key` este obligatoriu pentru elementele generate dintr-un tablou. El permite lui React să identifice eficient ce element s-a schimbat, s-a adăugat sau s-a șters între două re-randări – fără el, React ar fi forțat să recreeze toate elementele la orice modificare. **Cheia trebuie să fie unică în cadrul listei și stabilă în timp** (de la o re-randare la alta). Valoarea naturală este id-ul resursei. Folosirea indexului din tablou (`{produse.map((p, i) => <li key={i}>...)}`) este descurajată pentru liste care se reordonează sau în care se inserează/șterg elemente, pentru că poate produce comportament greșit: React „vede” că elementul de pe poziția 0 are aceeași cheie și nu observă că, de fapt, e alt element. [7, 28]

7. Props – datele primite din exterior

O componentă fără *props* este de utilitate limitată: produce mereu același rezultat. **Props** (prescurtarea de la „properties”) sunt argumentele unei componente, transmise în

momentul în care componenta este instanțiată în JSX. Sintactic, ele apar ca atribute pe tag-ul componentei:

```
<Salut nume="Ana" varsta={25} esteAdmin={true} />
```

În interiorul componentei, props sunt primite ca un singur argument – un obiect ale cărui chei sunt numele atributelor:

```
function Salut(props) {
  return (
    <p>
      Salut, {props.nume}! Ai {props.varsta} de ani.
      {props.esteAdmin && " Ești administrator."}
    </p>
  );
}
```

7.1. Destructurare în lista de parametri

Sintaxa de mai sus, deși corectă, este descurajată. În practică, se preferă destructurarea direct în lista de parametri:

```
function Salut({ nume, varsta, esteAdmin }) {
  return (
    <p>
      Salut, {nume}! Ai {varsta} de ani.
      {esteAdmin && " Ești administrator."}
    </p>
  );
}
```

Destructurarea face explicit, în semnătura funcției, ce props așteaptă componenta. Este o formă de auto-documentare și se folosește în aproape toate componentele scrise în codul real. [7, 15, 29, 30]

7.2. Valori implicite pentru props

Dacă o prop nu este transmisă, valoarea ei este undefined. Pentru a furniza o valoare implicită, se folosește sintaxa standard de destructurare cu valoare:

```
function Salut({ nume = "anonim", varsta = 0 }) {
  return <p>Salut, {nume}! Ai {varsta} de ani.</p>;
}

// Folosire:
<Salut />                  // "Salut, anonim! Ai 0 de ani."
<Salut nume="Ana" />       // "Salut, Ana! Ai 0 de ani."
```

7.3. Imutabilitatea props

Props sunt **imutabile din interiorul componentei**. Aceasta este o regulă strictă a React-ului: o componentă nu trebuie să își modifice propriile props. Cod ca cel de mai jos este o eroare:

```
function Salut({ nume }) {
  nume = "altcineva";  // GREȘIT - nu modifica props
  return <p>Salut, {nume}!</p>;
}
```

Dacă o componentă are nevoie de o variabilă pe care o modifică, aceasta este state, nu props. Distincția dintre cele două este fundamentală: props vin din afară (din părintele componentei) și sunt fixe pe durata unei randări; state este local componentei și poate fi modificat de aceasta. [7]

7.4. Prop-ul special: children

Există o prop cu rol special: **`children`**. Ea reprezintă conținutul JSX inserat între tag-ul de deschidere și cel de închidere al componentei:

```
function Card({ titlu, children }) {
  return (
    <div className="card">
      <h2>{titlu}</h2>
      <div className="card-body">{children}</div>
    </div>
  );
}

// Folosire:
<Card titlu="Detalii">
  <p>Acesta este conținutul cardului.</p>
  <button>Click</button>
</Card>
```

Această tehnică, numită *compoziție prin children*, este foarte folosită pentru componente de tip wrapper – cardurile, dialogurile modale, layout-urile. Componenta `Card` nu trebuie să cunoască ce conținut va primi; oferă doar un cadru vizual pentru orice. [7]

7.5. Primul exemplu cu props: ProductCard

Pentru a anticipa aplicația finală, definim o componentă ProductCard care primește un produs prin props și îl afișează:

```
// src/ProductCard.jsx
function ProductCard({ product }) {
  return (
    <div className="product-card">
      <h3>{product.name}</h3>
      <p>Preț: {product.price.toFixed(2)} lei</p>
      <p>Stoc: {product.stock} buc</p>
```

```
    </div>
  );
}

export default ProductCard;
```

Folosirea în App.jsx:

```
import ProductCard from "./ProductCard.jsx";
import "./App.css";

function App() {
  const produs = { id: 1, name: "Tastatură", price: 199.90, stock: 12 };
  return (
    <div>
      <h1>Catalog produse</h1>
      <ProductCard product={produs} />
    </div>
  );
}

export default App;
```

Observăm că prop-ul `product` este un obiect, nu un șir. Atunci când valoarea unei prop nu este un șir literal, se folosesc acolade: `product={produs}`. Doar valorile string pot fi date direct între ghilimele (`name="Tastatură"`). Toate celelalte valori (numere, booleani, obiecte, tablouri, funcții) trec prin acolade.

7.6. State – memoria locală a componentei

Props sunt date care vin din afara componentei. **State** sunt date pe care componenta le ține pentru ea însăși și pe care le modifică în timp. Diferența practică este crucială: când state se schimbă, React re-randează automat componenta, iar UI-ul reflectă noua valoare. Aceasta este buca de aur a programării reactive. [7, 31]

Mecanismul standard prin care o componentă declară state este hook-ul **`useState`**. *Hook-urile* sunt funcții speciale ale React-ului – numele lor începe cu **`use`** și ele „cuplează” componenta funcțională la mecanismele interne ale bibliotecii (state, efecte, context, etc.). Există un set fix de hook-uri standard, dintre care `useState` este cel mai folosit.

7.6.1. Anatomia unui useState

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Ai apăsat de {count} ori.</p>
      <button onClick={() => setCount(count + 1)}>
        Incrementează
```

```
        </button>
      </div>
    );
  }

  export default Counter;
```

Apelul `useState(0)` returnează un tablou de două elemente: valoarea curentă a stării (inițial `0`) și o funcție de actualizare. Prin destructurare, le primim ca `[count, setCount]`. Convenția de denumire este `[x, setX]` – numele variabilei și al setter-ului, cu prefixul `set` urmat de același nume scris cu majusculă.

Atunci când se apelează `setCount(noua_valoare)`, React marchează componenta ca fiind „modificată” și o programează pentru re-randare. La următorul tick al ciclului React, funcția `Counter` este apelată din nou; `useState` returnează noua valoare a stării; UI-ul se actualizează.

7.7. Greșeala clasică: mutarea directă a stării

Un începător este tentat să scrie:

```
<button onClick={() => count++}>Incrementează</button>          // GREȘIT
<button onClick={() => count = count + 1}>Incrementează</button>  // GREȘIT
```

Niciuna dintre aceste forme nu funcționează. Variabila `count` este `const` (nu o putem reasigna), iar chiar dacă am putea, React nu ar avea cum să afle că s-a schimbat și nu ar declanșa re-randarea. **Singura modalitate validă de a modifica state este apelul setter-ului**, aici `setCount(...)`.

7.8. Actualizări care depind de starea anterioară

În exemplul `Counter`, scrierea `setCount(count + 1)` funcționează în majoritatea cazurilor. Există însă un caz subtil în care nu funcționează corect: dacă setter-ul este apelat de mai multe ori într-o succesiune rapidă, valoarea `count` citită este aceeași în toate apelurile (cea din randarea curentă), nu cea actualizată după primul apel. [7]

```
// Vrem să incrementăm de 3 ori, dar...
const handleTripleClick = () => {
  setCount(count + 1);   // count este 0, deci setCount(1)
  setCount(count + 1);   // count este tot 0, deci setCount(1) din nou
  setCount(count + 1);   // count este tot 0, deci setCount(1) din nou
};
// Rezultat: count devine 1, nu 3.
```

Soluția este forma *funcțională* a setter-ului: în loc să-i pasăm o valoare, îi pasăm o funcție care primește valoarea curentă și returnează noua valoare:

```
const handleTripleClick = () => {
  setCount(c => c + 1);
  setCount(c => c + 1);
  setCount(c => c + 1);
```

```
};
// Rezultat: count devine 3. Corect.
```

Recomandarea practică: **atunci când noua valoare a stării depinde de cea anterioară, folosiți forma funcțională**. În cazul exemplului inițial `Counter`, forma `setCount(c => c + 1)` este preferabilă lui `setCount(count + 1)`.

7.9. Multiple variabile de stare

O componentă poate apela `useState` de mai multe ori, pentru a declara mai multe variabile de stare independente:

```
function Profil() {
  const [nume,  setNume]  = useState("");
  const [varsta, setVarsta] = useState(0);
  const [activ, setActiv] = useState(false);

  return (
    <div>
      <input value={nume} onChange={e => setNume(e.target.value)} />
      <input type="number" value={varsta}
             onChange={e => setVarsta(Number(e.target.value))} />
      <label>
        <input type="checkbox" checked={activ}
               onChange={e => setActiv(e.target.checked)} />
        Cont activ
      </label>
    </div>
  );
}
```

Alternativ, se poate folosi un singur `useState` cu un obiect:

```
const [profil, setProfil] = useState({ nume: "", varsta: 0, activ: false
});

// Modificare:
setProfil({ ...profil, nume: "Ana" });
```

Atenție: `useState` *înlocuiește integral* valoarea anterioară, nu o combină. De aceea, în exemplul de mai sus, este obligatoriu `{ ...profil, nume: "Ana" }` – așa preluăm celelalte câmpuri și suprascriem doar `nume`. Fără spread, `varsta` și `activ` ar dispărea.

7.10. Reguli pentru hook-uri

Hook-urile React respectă două reguli fundamentale care, dacă sunt încălcate, duc la comportament imprevizibil:

Apelați hook-urile doar la nivelul superior al unei componente. Nu apelați hook-uri în interiorul unor structuri `if`, `for`, funcții imbricate sau după o întoarcere timpurie. React se bazează pe ordinea fixă a apelurilor pentru a asocia fiecare `useState` cu „slot-ul” lui de stare. [7]

Apelați hook-urile doar din componente React sau din alte hook-uri. Nu apelați `useState` dintr-o funcție utilitară obișnuită – nu va funcționa.

ESLint, configurat de Vite, conține un plugin (`eslint-plugin-react-hooks`) care semnalează automat încălcarea acestor reguli. Erorile apar ca subliniere roșie în VS Code.

8. Tratarea evenimentelor

Evenimentele DOM (click, change, submit, focus, blur, keydown etc.) se atașează în JSX prin atribute speciale, scrise în *camelCase*: `onClick`, `onChange`, `onSubmit`, `onFocus`, `onBlur`, `onKeyDown`. Valoarea atributului este, **obligatoriu, o funcție** (nu un șir de cod, ca în HTML clasic):

```
// HTML clasic (NU FUNCȚIONEAZĂ în JSX):
<button onclick="alert('Click!')">Apasă</button>

// JSX corect:
<button onClick={() => alert("Click!")}>Apasă</button>
```

8.1. Argumentul evenimentului

Funcția primește un argument – un obiect numit *SyntheticEvent*, o variantă cross-browser a evenimentului DOM nativ. Cele mai folosite proprietăți ale lui sunt [7]:

- **`e.target`** – elementul care a declanșat evenimentul. Pentru un input, `e.target.value` conține valoarea curentă; pentru un checkbox, `e.target.checked` (un boolean).
- **`e.preventDefault()`** – oprește comportamentul implicit al browserului. Folosit cel mai des la submit-ul de formular (pentru a evita reîncărcarea paginii) și la click pe `<a>` (pentru a evita navigarea).
- **`e.stopPropagation()`** – oprește propagarea evenimentului în arborele DOM. Util în situații rare (de exemplu, un click pe un buton care nu trebuie să declanșeze și click-ul pe containerul care îl conține).
- **`e.key`** – pentru `onKeyDown`, numele tastei apăsate (ex. `"Enter"`, `"Escape"`, `"a"`).

8.2. Pattern-uri uzuale

8.2.1. Click pe buton, fără argumente

```
<button onClick={() => setCount(count + 1)}>+1</button>
```

8.2.2. Click pe buton, cu argumente

Atunci când vrem să transmitem date suplimentare către handler, le încadrăm într-o funcție anonimă:

```
{produse.map(p => (
  <button key={p.id} onClick={() => stergeProdus(p.id)}>
    Șterge {p.name}
  </button>
```

```
))}
```

Atenție: a scrie `onClick={stergeProdus(p.id)}` este o greșeală frecventă – aici am *apela* funcția pe loc, nu am trimite-o ca handler. Diferența: `onClick` primește o funcție pe care o va apela *ulterior*, când are loc click-ul. [7, 11]

8.2.3. Schimbarea unui input

```
const [nume, setNume] = useState("");

<input
  type="text"
  value={nume}
  onChange={e => setNume(e.target.value)}
/>
```

Setarea `value={nume}` transformă input-ul într-un *input controlat*: valoarea afișată este mereu cea din state, iar la fiecare tastă, `onChange` actualizează state-ul.

8.2.4. Submit de formular

```
function Formular() {
  const [nume, setNume] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();           // OBLIGATORIU
    console.log("Trimis:", nume);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input value={nume} onChange={e => setNume(e.target.value)} />
      <button type="submit">Trimite</button>
    </form>
  );
}
```

Apelul `e.preventDefault()` este obligatoriu, altfel browser-ul va încerca să trimită formularul către URL-ul curent (printr-un GET) și va recarca pagina, distrugând starea aplicației SPA.

9. Compoziția componentelor

O aplicație React reală nu este o singură componentă, ci un arbore de componente, fiecare având o responsabilitate clară. Compoziția este principalul instrument de organizare – analog descompunerii unei probleme în funcții, dar la nivelul UI-ului. [30, 32]

9.1. Date în jos, evenimente în sus

Modelul standard de comunicare în React este *data down, events up* (date în jos, evenimente în sus). Componenta părinte transmite date copiilor prin props; copiii notifică părintele despre acțiuni prin callback-uri (funcții transmise tot ca props). [30]

Iată un exemplu complet, cu trei componente:

```
// src/StockBadge.jsx
function StockBadge({ stock }) {
  const className = stock > 0 ? "in-stock" : "out-of-stock";
  const text = stock > 0 ? "În stoc" : "Indisponibil";
  return <span className={className}>{text}</span>;
}

export default StockBadge;


// src/ProductCard.jsx
import StockBadge from "./StockBadge.jsx";

function ProductCard({ product, onSelect }) {
  return (
    <div className="product-card">
      <h3>{product.name}</h3>
      <p>Preț: {product.price.toFixed(2)} lei</p>
      <p>
        Stoc: {product.stock}
        <StockBadge stock={product.stock} />
      </p>
      <button onClick={() => onSelect(product)}>Selectează</button>
    </div>
  );
}

export default ProductCard;

// src/App.jsx
import { useState } from "react";
import ProductCard from "./ProductCard.jsx";
import "./App.css";

const PRODUSE = [
  { id: 1, name: "Tastatură", price: 199.90, stock: 12 },
  { id: 2, name: "Monitor",   price: 1299.0, stock:  3 },
  { id: 3, name: "Mouse",     price:  79.50, stock:  0 },
];

function App() {
  const [selectat, setSelectat] = useState(null);

  return (
    <div className="app">
      <h1>Catalog produse</h1>
```

```
        <div className="product-list">
          {PRODUSE.map(p => (
            <ProductCard
              key={p.id}
              product={p}
              onSelect={setSelectat}
            />
          ))}
        </div>

        {selectat && (
          <div className="selected-info">
            Produs selectat: <strong>{selectat.name}</strong>
          </div>
        )}
      </div>
    );
  }

  export default App;
```

Să analizăm fluxul de date și de evenimente:

Datele coboară: `App` ține lista de produse și starea „produs selectat”. Pentru fiecare produs, transmite obiectul `product` către `ProductCard`. `ProductCard` transmite `stock` către `StockBadge`. Fiecare componentă primește exact ce-i trebuie pentru a-și face treaba.

Evenimentele urcă: când utilizatorul apasă butonul „Selectează” din `ProductCard`, componenta apelează callback-ul primit prin props (`onSelect(product)`). Acest callback este, de fapt, `setSelectat` din `App`. Astfel, `App` află despre eveniment și își actualizează propria stare. Re-randarea se propagă în jos.

9.2. Lifting state up

În exemplul de mai sus, starea „produs selectat” rezidă în `App`, nu în `ProductCard`. Aceasta este o aplicație a principiului *lifting state up*: când mai multe componente au nevoie să cunoască sau să modifice o aceeași informație, **starea se mută la cel mai apropiat strămoș comun** care le conține pe toate. Dacă starea ar fi în interiorul lui `ProductCard`, fiecare card ar fi avut propria sa noțiune de „selectat”, ceea ce nu corespunde modelului real (un singur produs poate fi selectat la un moment dat). [7]

10. Componente de prezentare vs componente cu logică

O distincție utilă în organizarea unei aplicații React este între:

Componente de prezentare („dumb components”) – cunosc doar despre cum se afișează ceva, nu și despre cum se obține informația. Primesc totul prin props. Exemple aici: `ProductCard`, `StockBadge`. Sunt ușor de reutilizat și de testat. [32, 45]

Componente cu logică („smart components”) – orchestrează datele, gestionează state-ul, comunică cu serverul. Exemplu aici: `App`. În aplicații mai mari, ele se numesc adesea „pagini” sau „containere”.

Această separare este o convenție utilă, nu o regulă rigidă. Tinde să producă cod mai curat, în care componentele de afișare sunt simple, reutilizabile și ușor de înțeles, iar logica de business este concentrată în câteva locuri identificabile.

11. Aplicația laboratorului: catalogul de produse cu date hardcodate

Punem împreună tot ce am învățat și construim aplicația finală a laboratorului. Scopul este să avem o componentă App funcțională, capabilă să afișeze o listă de produse hardcodate, cu interactivitate minimă (selectarea unui produs), folosind componente reutilizabile. În Laboratorul 11, vom înlocui array-ul hardcodat cu un apel către API-ul REST – tot restul rămâne la fel.

11.1. Structura proiectului

```
src/
├── App.css
├── App.jsx
├── index.css
├── main.jsx
├── ProductCard.jsx
└── StockBadge.jsx
```

11.2. Codul complet

Fișierul `src/StockBadge.jsx`:

```
function StockBadge({ stock }) {
  const isAvailable = stock > 0;
  const className = "stock-badge " +
    (isAvailable ? "stock-in" : "stock-out");
  const text = isAvailable ? "În stoc" : "Indisponibil";
  return <span className={className}>{text}</span>;
}

export default StockBadge;
```

Fișierul `src/ProductCard.jsx`:

```
import StockBadge from "./StockBadge.jsx";

function ProductCard({ product, onSelect }) {
  return (
    <div className="product-card">
      <h3>{product.name}</h3>
      <div className="product-price">
        {product.price.toFixed(2)} lei
      </div>
```

```
        <div className="product-stock">
          Stoc: {product.stock} buc
          <StockBadge stock={product.stock} />
        </div>
        <button
          className="btn-select"
          onClick={() => onSelect(product)}
          disabled={product.stock === 0}
        >
          Selectează
        </button>
      </div>
    );
  }

  export default ProductCard;
```

Fișierul `src/App.jsx`:

```
  import { useState } from "react";
  import ProductCard from "./ProductCard.jsx";
  import "./App.css";

  const PRODUSE_INITIALE = [
    { id: 1, name: "Tastatură mecanică", price: 349.90, stock: 12 },
    { id: 2, name: "Monitor 27 inch 4K", price: 1899.0, stock:  3 },
    { id: 3, name: "Mouse wireless",     price: 159.50, stock:  0 },
    { id: 4, name: "Webcam HD",          price: 229.00, stock:  7 },
  ];

  function App() {
    const [produse]   = useState(PRODUSE_INITIALE);
    const [selectat,  setSelectat]  = useState(null);
    const [filtru,    setFiltru]    = useState("");

    const produseFiltrate = produse.filter(p =>
      p.name.toLowerCase().includes(filtru.toLowerCase())
    );

    return (
      <div className="app">
        <header>
          <h1>Catalog produse</h1>
          <input
            className="search"
            placeholder="Caută..."
            value={filtru}
            onChange={e => setFiltru(e.target.value)}
          />
        </header>

        <main>
          <section className="product-list">
```

```
            {produseFiltrate.length === 0
              ? <p className="empty">Niciun produs găsit.</p>
              : produseFiltrate.map(p => (
                  <ProductCard
                    key={p.id}
                    product={p}
                    onSelect={setSelectat}
                  />
                ))}
          </section>

          {selectat && (
            <aside className="selected-info">
              <h2>Produs selectat</h2>
              <p><strong>{selectat.name}</strong></p>
              <p>Preț: {selectat.price.toFixed(2)} lei</p>
              <p>Stoc: {selectat.stock} buc</p>
              <button onClick={() => setSelectat(null)}>
                Închide
              </button>
            </aside>
          )}
        </main>
      </div>
    );
}

export default App;
```

Fișierul `src/App.css` conține un set minim de stiluri:

```
.app {
  max-width: 900px;
  margin: 0 auto;
  padding: 16px;
  font-family: system-ui, sans-serif;
}

header {
  display: flex;
  justify-content: space-between;
  align-items: center;
  margin-bottom: 24px;
}

.search {
  padding: 8px 12px;
  border: 1px solid #ccc;
  border-radius: 4px;
  font-size: 14px;
  width: 240px;
}

main {
  display: grid;
```

```
    grid-template-columns: 1fr 280px;
    gap: 24px;
  }

  .product-list {
    display: grid;
    grid-template-columns: repeat(auto-fill, minmax(220px, 1fr));
    gap: 16px;
  }

  .product-card {
    border: 1px solid #ddd;
    border-radius: 6px;
    padding: 16px;
    background: white;
  }

  .product-price {
    font-size: 20px;
    font-weight: bold;
    color: #2c5282;
    margin: 8px 0;
  }

  .product-stock {
    font-size: 14px;
    color: #555;
    margin-bottom: 12px;
  }

  .stock-badge {
    display: inline-block;
    padding: 2px 8px;
    border-radius: 12px;
    font-size: 12px;
    margin-left: 8px;
  }

  .stock-in  { background: #c6f6d5; color: #22543d; }
  .stock-out { background: #fed7d7; color: #742a2a; }

  .btn-select {
    background: #2c5282;
    color: white;
    border: none;
    padding: 8px 16px;
    border-radius: 4px;
    cursor: pointer;
  }

  .btn-select:disabled {
    background: #a0aec0;
    cursor: not-allowed;
  }
```

```
.selected-info {
  border: 1px solid #ddd;
  border-radius: 6px;
  padding: 16px;
  background: #f7fafc;
  height: fit-content;
}

.empty {
  color: #888;
  font-style: italic;
}
```

11.3. Câteva cuvinte despre stilizare

În exemplul de mai sus, s-a folosit CSS clasic, importat în componentă cu `import "./App.css"`. Acesta este modul cel mai simplu și suficient pentru aplicații de dimensiune medie. Se cuvin câteva precizări [11, 46]:

- Importul CSS este *global*: clasele definite în `App.css` sunt vizibile în toată aplicația. Acest lucru poate duce la conflicte de nume în aplicații mari.
- Pentru a evita conflictele, există **`CSS Modules`**: fișierele cu numele `Foo.module.css` produc clase scoped (numele este transformat automat astfel încât să fie unic la nivelul aplicației).
- Alternative moderne: **Tailwind CSS** (clase utilitare predefinite), **styled-components** sau **Emotion** (CSS-in-JS), **shadcn/ui** (set de componente pre-construite peste Tailwind).

În cadrul acestui îndrumar se va folosi CSS clasic.

12. Concluzii

Acest laborator a parcurs toate mecanismele fundamentale ale React-ului modern: definirea unei componente ca funcție, sintaxa JSX și diferențele față de HTML, transmiterea datelor prin props, gestionarea stării locale prin useState, tratarea evenimentelor și compoziția componentelor.

Esențialul de reținut, înainte de a continua:

- componentă React este o funcție care primește props și returnează JSX.
- JSX este transformat în apeluri `React.createElement` – arată ca HTML, dar este JavaScript.
- Props sunt date care vin din afară, sunt imutabile, și se folosesc pentru a parametriza o componentă.
- State sunt date locale ale componentei, modificate prin setter-ul lui `useState`. Modificarea declanșează re-randarea.
- Comunicarea în arborele de componente respectă modelul „date în jos, evenimente în sus”.

- Listele se randează cu `.map()` și necesită un atribut `key` unic și stabil.

13. Exerciții propuse

13.1. Exercițiul 1 – Componenta Pret

Extrageți afișarea prețului dintr-un `ProductCard` într-o componentă separată `Pret`, care primește două props: `valoare` (numărul) și `moneda` (un șir, implicit "lei"). Componenta formatează prețul cu două zecimale și include unitatea monetară. Verificați că componenta funcționează atât cu `<Pret valoare={199.9} />` cât și cu `<Pret valoare={50} moneda="EUR" />`.

13.2. Exercițiul 2 – Sortare

Adăugați în `App` un selector (`<select>`) cu trei opțiuni: "Implicit", "Preț crescător", "Preț descrescător". O nouă variabilă de stare `sortBy` ține opțiunea curentă. Lista de produse afișată este sortată conform acestei opțiuni.

Atenție: nu modificați direct tabloul `produse` (nu apelați `.sort()` pe el). Creați mai întâi o copie cu `[...produse]` și sortați copia. Această disciplină este aliniată cu principiul mai general al React-ului de a nu modifica state-ul direct.

13.3. Exercițiul 3 – Componenta ProductSummary

Construiți o componentă `ProductSummary` care primește prin props lista de produse și afișează: numărul total de produse, valoarea totală a stocului (suma `price * stock` pentru fiecare produs), prețul mediu, și numărul de produse indisponibile (cu stoc 0). Componenta nu are state propriu, doar calculează și afișează (componentă derivată).

Folosiți componenta `ProductSummary` în `App`, deasupra listei. Pentru calcule, folosiți `Array.prototype.reduce` sau `Array.prototype.filter` după caz.

13.4. Exercițiul 4 – Editarea stocului

Extindeți `App` astfel încât pe fiecare `ProductCard` să existe două butoane mici: "+" și "−". Click pe "+" crește stocul produsului cu 1, click pe "−" îl scade cu 1 (dar nu sub 0).

Pentru a realiza acest lucru, va trebui să mutați state-ul `produse` într-un `useState` modificabil (acum este `const [produse]` – fără setter, pentru că este doar inițial). Veți avea nevoie de o funcție `actualizeazaStoc(productId, delta)` în `App`, care produce un array nou de produse cu stocul modificat pentru produsul cu id-ul respectiv. Transmiteți această funcție către ProductCard prin props.

Indicație: pentru a actualiza un singur element într-un array imutabil, modelul este `produse.map(p => p.id === productId ? {...p, stock: p.stock + delta} : p)`. Acest tipar este foarte des folosit în React.

Laborator 8: Conectarea la API-ul REST. useEffect, axios și afișarea listei de produse

1. Obiectivul laboratorului

Acest laborator marchează prima întâlnire reală dintre frontend-ul React construit în laboratorul precedent și backend-ul Spring Boot construit în laboratoarele anterioare. Până în acest moment, aplicația React afișa o listă de produse hardcodate, declarate static într-o constantă în interiorul componentei App. La finalul laboratorului, această listă va fi încărcată dinamic, prin HTTP, de pe endpoint-ul `/api/products` al aplicației Spring Boot.

Pe parcurs, vom întâlni două probleme noi care nu au apărut în laboratorul precedent. Prima este legată de comunicarea între două origini diferite (backend pe portul 8080, frontend pe portul 5173) și ne va obliga să configurăm CORS[30] pe backend. A doua este legată de momentul în care declanșăm cererea HTTP – nu o putem pune direct în corpul unei componente, pentru că ar produce cereri repetate la fiecare re-randare; vom învăța hook-ul `useEffect`, mecanismul React pentru efecte secundare. Mai învățăm să tratăm corect cele patru stări vizibile ale unei componente conectate la server: încărcare, eroare, listă goală, listă populată.

La finalul laboratorului, cititorul ar trebui să fie capabil:

- să explice politica *Same-Origin* a browserelor și mecanismul CORS care o relaxează controlat;
- să configureze CORS într-o aplicație Spring Boot, atât global, cât și la nivel de controller individual;
- să instaleze și să folosească biblioteca `axios` pentru a face cereri HTTP din JavaScript;
- să utilizeze hook-ul `useEffect` pentru a executa efecte secundare la momentul potrivit din ciclul de viață al unei componente;
- să explice rolul array-ului de dependențe al `useEffect` și să recunoască greșelile uzuale (efect care nu rulează niciodată, ciclu infinit de re-randări);
- să gestioneze corect cele patru stări vizibile ale unei componente conectate la server (loading, error, empty, populated);

[30] CORS (Cross-Origin Resource Sharing) este un mecanism de securitate din browsere care controlează dacă o aplicație web poate face cereri HTTP către un server aflat pe un alt domeniu, port sau protocol. Specificația CORS este disponibilă la https://fetch.spec.whatwg.org/ ca parte a *Fetch Standard*, iar discuția istorică originală la https://www.w3.org/TR/cors/.

- să separe codul de comunicare HTTP într-un modul dedicat și să motiveze această separare arhitecturală.

2. Modelul comunicării într-o aplicație SPA

O aplicație SPA, odată încărcată în browser, devine independentă de server în privința navigării: nu mai există cereri pentru pagini noi. Singura comunicare cu serverul este sub formă de cereri HTTP către API: fie pentru a citi date (`GET`), fie pentru a le modifica (`POST`, `PUT`, `DELETE`). Răspunsurile sunt în format JSON, care în JavaScript se transformă natural în obiecte. [18, 40]

Două lumi separate. În timpul dezvoltării, backend-ul Spring Boot rulează pe `http://localhost:8080` (portul implicit Spring Boot), iar aplicația React, pornită cu `npm run dev`, rulează pe `http://localhost:5173` (portul implicit Vite). Pentru browser, acestea sunt *două origini diferite* – iar aici intervine o problemă fundamentală de securitate.

Pentru a fixa terminologia, o origine este definită ca un triplet:

```
origine = (schema, host, port)

http://localhost:5173        // origine A
http://localhost:8080        // origine B  (port diferit)
https://localhost:5173       // origine C  (schemă diferită)
http://example.com:5173      // origine D  (host diferit)
```

Două URL-uri aparțin aceleiași origini doar dacă au aceeași schemă (http vs https), același host (numele de domeniu sau adresa IP) și același port. Dacă măcar una dintre cele trei diferă, sunt origini distincte.

3. Same-Origin Policy și CORS

3.1. Same-Origin Policy

Same-Origin Policy (SOP) este una dintre pietrele de temelie ale securității webului. Implementată de toate browserele moderne, ea interzice unei pagini web să facă cereri AJAX către o origine diferită de cea de pe care a fost servită. Politica datează din 1995, de la *Netscape Navigator*, și a fost rafinată de-a lungul anilor. [11, 17]

Motivul existenței SOP este o categorie întreagă de atacuri pe care le-ar permite absența ei. Putem considera următorul scenariu: un utilizator este autentificat în aplicația de internet banking a băncii sale (cookie-uri de sesiune setate pe domeniul băncii). Într-un alt tab, deschide un site oarecare. Fără SOP, codul JavaScript al acelui site ar putea face liber cereri către domeniul băncii, transportând automat cookie-urile de sesiune ale utilizatorului – și astfel ar putea citi soldul, transfera bani, schimba date de contact. SOP împiedică acest scenariu: chiar dacă browserul are cookie-urile, codul JavaScript de pe alt domeniu nu poate citi răspunsurile cererilor făcute către domeniul băncii.

Mai precis, SOP nu blochează cererea în sine (browserul o trimite), ci *blochează codul JavaScript să citească răspunsul*. Acest detaliu este uneori sursa de confuzie când se investighează probleme de CORS.

3.2. CORS – Cross-Origin Resource Sharing

SOP, deși necesar din motive de securitate, este uneori prea restrictiv. Există situații legitime în care un frontend trebuie să comunice cu un backend pe altă origine: aplicații SPA cu backend separat (cazul nostru), API-uri publice consumate de pe site-uri terțe, microservicii distribuite pe mai multe domenii. CORS (*Cross-Origin Resource Sharing*), standardizat de W3C[31] în 2014 și rafinat de WHATWG[32], este mecanismul prin care un server poate *explicit* să permită anumitor origini să facă cereri către el și să citească răspunsurile. [17]

Tehnic, CORS funcționează prin antete HTTP. Atunci când browserul detectează o cerere *cross-origin* (frontend de pe `localhost:5173` către backend pe `localhost:8080`), poate avea două comportamente, în funcție de tipul și caracteristicile cererii.

3.2.1. Cereri simple

Pentru cererile considerate „simple” (`GET` sau `POST` cu antete standard, fără credențiale speciale), browserul trimite cererea direct, dar adaugă un antet:

```
Origin: http://localhost:5173
```

Serverul, atunci când răspunde, include în răspuns antetul `Access-Control-Allow-Origin` care indică ce origini sunt permise:

```
Access-Control-Allow-Origin: http://localhost:5173
```

Dacă valoarea acestui antet include originea cererii (sau este `*`, însemnând „orice origine”), browserul livrează răspunsul codului JavaScript. Altfel, blochează accesul la răspuns și aruncă o eroare.

3.2.2. Cereri preflight

Pentru cererile non-triviale (`PUT`, `DELETE`, `PATCH`, sau `GET/POST` cu antete personalizate, sau `Content-Type: application/json`), browserul folosește un mecanism numit *preflight*. Înainte de cererea efectivă, trimite o cerere `OPTIONS` care „întreabă” serverul dacă cererea propriu-zisă este permisă:

```
OPTIONS /api/products HTTP/1.1
Origin: http://localhost:5173
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: content-type
```

[31] World Wide Web Consortium (W3C) este organizația internațională care dezvoltă și standardizează tehnologiile fundamentale ale web-ului, precum HTML, CSS și XML.
[32] WHATWG este o organizație care dezvoltă și menține standarde moderne pentru tehnologiile web, în special standardul HTML folosit de browserele moderne.

Serverul răspunde cu permisiunile aplicabile:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin:  http://localhost:5173
Access-Control-Allow-Methods: GET, POST, PUT, DELETE
Access-Control-Allow-Headers: Content-Type
Access-Control-Max-Age:       3600
```

Doar dacă răspunsul preflight este pozitiv, browserul trimite cererea propriu-zisă (`PUT`-ul, în exemplul de mai sus). Antetul `Access-Control-Max-Age` permite cache-uirea rezultatului preflight pentru un anumit număr de secunde, evitând o cerere `OPTIONS` pentru fiecare apel.

3.2.3. Eroarea tipică în consolă

Un frontend care nu este permis explicit de backend produce, în consola browserului, o eroare caracteristică:

```
Access to XMLHttpRequest at 'http://localhost:8080/api/products'
from origin 'http://localhost:5173' has been blocked by CORS
policy: No 'Access-Control-Allow-Origin' header is present
on the requested resource
```

Această eroare nu vine de la serverul Spring Boot (în log-urile lui, cererea apare ca succes, cu cod 200), ci de la *browserul însuși*, care a primit răspunsul, a constatat absența antetelor CORS și a refuzat să-l livreze codului JavaScript. Confuzia este alimentată de faptul că, dacă deschidem direct `http://localhost:8080/api/products` în browser, vedem perfect JSON-ul – pentru că, în acel caz, originea paginii și a cererii sunt aceleași.

4. Configurarea CORS în Spring Boot

Spring Boot oferă mai multe modalități de a configura CORS, cu niveluri diferite de granularitate. Le prezentăm pe cele trei mai uzuale, în ordinea sferei de aplicare. [2, 17]

4.1. Varianta 1: adnotarea @CrossOrigin la nivel de controller

Cea mai simplă variantă este adnotarea `@CrossOrigin` aplicată direct pe clasa controller sau pe o metodă. Adăugăm în `ProductController`:

```
@RestController
@RequestMapping("/api/products")
@CrossOrigin(origins = "http://localhost:5173")
public class ProductController {
    // ... metodele existente
}
```

Această adnotare este utilă pentru prototipuri rapide sau pentru endpoint-uri izolate. Dezavantajul ei major este *duplicarea*: dacă există mai multe controllere, va trebui repetată pe fiecare. De asemenea, configurarea trăiește răspândită în cod, ceea ce face mai greu de auditat ce origini sunt permise în întreaga aplicație.

4.2. Varianta 2: configurare globală prin WebMvcConfigurer

Pentru aplicații reale, este preferabil să se configureze CORS într-un singur loc, central. Aceasta se face prin implementarea interfeței `WebMvcConfigurer` și suprascrierea metodei `addCorsMappings`. Creăm o nouă clasă în pachetul de configurare al aplicației Spring Boot:

```
// src/main/java/ro/unitbv/restlab/config/CorsConfig.java

package ro.unitbv.restlab.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.*;

@Configuration
public class CorsConfig implements WebMvcConfigurer {

  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
        .allowedOrigins("http://localhost:5173")
        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
        .allowedHeaders("*")
        .maxAge(3600);
  }
}
```

Analiza metodei:

- `addMapping("/api/**")` – aplică această configurare CORS la toate path-urile care încep cu `/api/`. Pattern-ul `**` înseamnă „orice succesiune de segmente”, deci sunt acoperite atât `/api/products`, cât și `/api/products/5` sau `/api/orders/123/items`.
- `allowedOrigins(...)` – lista originilor permise. În dezvoltare, este `http://localhost:5173` (Vite). Se pot trece mai multe origini, separate prin virgulă, sau se poate folosi `*` pentru orice origine (nerecomandat în producție).
- `allowedMethods(...)` – metodele HTTP permise. Includem explicit `OPTIONS` pentru cererile preflight.
- `allowedHeaders("*")` – orice antet de cerere este permis. În producție, ar fi mai prudent să se enumere explicit doar antetele necesare (de exemplu, `Content-Type`, `Authorization`).
- `maxAge(3600)` – numărul de secunde pentru care browserul poate cache-ui răspunsul preflight (în exemplu este o oră). Reduce numărul de cereri `OPTIONS` suplimentare.

> **Atenție**
>
> Combinația `allowedOrigins("*")` împreună cu `allowCredentials(true)` este *interzisă* de specificația CORS. Browserul va respinge orice astfel de configurare. Dacă

> aplicația are nevoie să trimită cookie-uri sau alte credențiale, originile trebuie enumerate explicit (sau trebuie folosit `allowedOriginPatterns`, care permite wildcards mai flexibile).

4.3. Varianta 3: configurare prin properties

O alternativă, utilă mai ales când dorim să schimbăm originile între medii (dezvoltare, staging, producție) fără recompilare, este să citim valorile dintr-un fișier `application.properties`:

```
# application.properties
app.cors.allowed-origins=http://localhost:5173,https://produse.example.com
```

Și în `CorsConfig`:

```
@Configuration
public class CorsConfig implements WebMvcConfigurer {

  @Value("${app.cors.allowed-origins}")
  private String[] allowedOrigins;

  @Override
  public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping("/api/**")
        .allowedOrigins(allowedOrigins)
        .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS")
        .allowedHeaders("*")
        .maxAge(3600);
  }
}
```

Această variantă este profesional preferată în aplicații reale. În cadrul laboratorului folosim varianta 2, mai compactă, urmând ca tehnica de externalizare a configurației să fie reluată într-un context mai general.

4.3.1. Verificarea configurării

După adăugarea clasei `CorsConfig`, se repornește aplicația Spring Boot. Verificarea se poate face direct cu `curl`, fără implicarea browserului, că antetele sunt corecte:

```
curl -i -X OPTIONS http://localhost:8080/api/products \
  -H "Origin: http://localhost:5173" \
  -H "Access-Control-Request-Method: GET"
```

Răspunsul așteptat conține antetele `Access-Control-Allow-*` corespunzătoare:

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://localhost:5173
Access-Control-Allow-Methods: GET, POST, PUT, DELETE, OPTIONS
Access-Control-Allow-Headers: Content-Type
Access-Control-Max-Age: 3600
...
```

Dacă lipsesc, este un semn că adnotarea `@Configuration` nu este preluată (pachetul clasei nu este scanat de Spring Boot) sau că aplicația nu a fost repornită.

5. Instrumentul de comunicare: axios

Pentru efectuarea cererilor HTTP din JavaScript există două opțiuni principale: API-ul nativ `fetch` (inclus în toate browserele moderne) și biblioteca `axios`. Tabelul următor sintetizează diferențele principale.

Aspect	*fetch*	*axios*
Disponibilitate	Nativ în browsere și Node 18+	Bibliotecă terță (`npm install axios`)
Parsare JSON	Manuală: `await res.json()`	Automată; `res.data` conține obiectul
Erori HTTP (4xx, 5xx)	Promisiunea reușește; trebuie verificat `res.ok`	Promisiunea cade pe ramura `.catch` automat
Configurare baseURL / antete implicite	Manual la fiecare apel	Prin `axios.create({...})`
Interceptori	Nu există nativ	Suportați nativ pentru cereri și răspunsuri
Mărime în bundle	0 KB (deja în browser)	≈ 13 KB minificat și gzip

Ambele alternative sunt complet funcționale; alegerea este, în mare măsură, o chestiune de preferință. În cadrul acestui îndrumar folosim `axios`, pentru trei motive: parsarea automată a JSON-ului face codul mai scurt și mai lizibil; tratarea uniformă a erorilor HTTP elimină verificarea explicită a `res.ok`; configurarea unei instanțe cu `baseURL` evită repetarea adresei serverului în fiecare apel. Toate aceste beneficii sunt vizibile încă de la primele exemple. [10, 17]

5.1. Instalare

Dintr-un terminal poziționat în directorul `products-ui` (proiectul React creat în laboratorul anterior), se rulează:

```
npm install axios
```

Comanda adaugă `axios` în `package.json` la secțiunea `dependencies` (este o dependență de runtime, nu doar de dezvoltare) și descarcă pachetul în `node_modules/`. Verificarea că instalarea a reușit:

```
// În package.json apare:
"dependencies": {
  "axios": "^1.7.0",
  "react": "^18.3.1",
  "react-dom": "^18.3.1"
}
```

5.2. Sintaxa de bază

Forma cea mai simplă a unui apel cu axios:

```
import axios from "axios";
```

```
const URL = "http://localhost:8080/api/products";

// Forma cu .then() / .catch():
axios.get(URL)
  .then(response => {
    console.log(response.data); // array-ul de produse
  })
  .catch(error => {
    console.error(error);
  });

// Forma cu async / await (echivalentă, mai citibilă):
async function loadProducts() {
  try {
    const response = await axios.get(URL);
    console.log(response.data);
  } catch (error) {
    console.error(error);
  }
}
```

Obiectul `response` returnat de axios conține mai multe câmpuri utile:

- `response.data` – corpul răspunsului, deja parsat din JSON. Pentru noi, este array-ul de produse.
- `response.status` – codul HTTP (200, 201, 404, ...).
- `response.statusText` – descrierea textuală ("OK", "Not Found", ...).
- `response.headers` – antetele HTTP ale răspunsului.

În cazul unei erori (cod 4xx sau 5xx), axios respinge promisiunea, iar obiectul `error` primit în catch are forma:

- `error.response` – obiectul răspuns primit de la server (există dacă cererea a ajuns și serverul a răspuns cu cod de eroare).
- `error.request` – cererea trimisă (există dacă cererea s-a trimis dar nu s-a primit răspuns – server picat, timeout, problemă de rețea).
- `error.message` – mesajul de eroare textual.

6. Hook-ul useEffect – efecte secundare

Avem axios instalat, CORS configurat. Suntem tentați să scriem direct, în interiorul componentei App, ceva de genul:

```
import axios from "axios";

function App() {

  // ATENȚIE: cod GREȘIT, doar ilustrativ

  const response = await axios.get("http://localhost:8080/api/products");
```

```
    const produse = response.data;
    return <div>{produse.map(...)}</div>;
  }
```

Acest cod nu funcționează din mai multe motive.

Primul motiv: corpul unei componente React trebuie să fie sincron. Nu putem folosi `await` în corpul direct al unei funcții componentă, decât dacă o transformăm în `async` – ceea ce React nu acceptă.

Al doilea motiv (mai profund): corpul unei componente trebuie să fie pur. React poate apela funcția unei componente de mai multe ori într-un interval scurt – pentru a calcula re-randări, pentru optimizări, pentru testarea efectelor în *Strict Mode*. Dacă fiecare apel ar declanșa o cerere HTTP, am inunda serverul cu cereri inutile. [7, 32]

6.1. Ce sunt efectele secundare

Un ***efect secundar*** este orice acțiune care iese din regimul pur al unei funcții: apel HTTP, modificarea unei variabile globale, scriere în `console.log`, interacțiune directă cu DOM-ul, abonare la un eveniment de browser, pornirea unui timer. Acestea nu trebuie să se afle în corpul direct al componentei. React oferă pentru ele un mecanism dedicat: hook-ul `useEffect`. [7]

6.2. Anatomia useEffect

```
  import { useEffect } from "react";

  function MyComponent() {
    useEffect(() => {
      // Codul efectului - rulează DUPĂ ce componenta a fost randată
      console.log("Componenta a fost randată");
    }, [/* array de dependențe */]);

    return <div>...</div>;
  }
```

`useEffect` primește două argumente:

- **Prima poziție** – o funcție care conține codul efectului (cererea HTTP, abonarea, etc.). Această funcție este executată *după* ce React a terminat de actualizat DOM-ul.
- **A doua poziție** – un tablou de dependențe. Acesta determină *când* se va rula efectul.

6.3. Array-ul de dependențe – cazurile fundamentale

6.3.1. Caz 1: array gol []

```
  useEffect(() => {
    console.log("Rulez o singură dată, după primul render");
  }, []);
```

Cu un array gol, efectul rulează exact o dată, după primul render. Aceasta este forma standard pentru *operații de inițializare*: încărcarea inițială a datelor, abonarea la un eveniment global, pornirea unui timer. Este cazul nostru de astăzi: vrem să încărcăm produsele o dată, la deschiderea componentei. [7]

6.3.2. Caz 2: array cu valori [x, y]

```
useEffect(() => {
  console.log("Rulez când se schimbă productId");
  loadProductDetails(productId);
}, [productId]);
```

Efectul rulează după primul render și *ori de câte ori se schimbă* vreuna dintre valorile din array. Forma este utilă, de exemplu, pentru a re-încărca datele când utilizatorul navighează către alt produs.

6.3.3. Caz 3: niciun tablou (atenție, capcană)

```
useEffect(() => {
  console.log("Rulez după FIECARE render");
});
```

Fără tablou de dependențe, efectul rulează după *fiecare* randare a componentei. Aceasta este, în 99% din cazuri, o **greșeală**. Combinat cu un apel HTTP care, la întoarcere, modifică starea (declanșând re-randarea), produce un *ciclu infinit* de cereri către server. Vom vedea exemplul concret puțin mai jos.

6.4. Funcția de cleanup

Funcția trimisă ca prim argument poate, opțional, să returneze o altă funcție – numită funcția de cleanup. Aceasta este apelată de React fie când componenta este demontată (scoasă din arbore), fie înainte de o nouă rulare a efectului (dacă dependențele s-au schimbat):

```
useEffect(() => {
  // Pornesc un timer
  const timerId = setInterval(() => {
    console.log("tick");
  }, 1000);

  // Cleanup: opresc timer-ul
  return () => {
    clearInterval(timerId);
  };
}, []);
```

Funcția de cleanup este esențială pentru a evita scurgerile de memorie cu timere, abonări la WebSocket-uri, evenimente globale, sau cereri HTTP în curs care nu mai trebuie procesate. [7]

6.5. Strict Mode și efectele rulate de două ori

În laboratorul anterior se observă că `main.jsx` conține componenta `<StrictMode>`. Una dintre consecințele ei vizibile este că, în modul de dezvoltare, fiecare `useEffect` este rulat de două ori la prima montare a componentei: setup → cleanup → setup. Aceasta nu este un bug, ci o verificare intenționată a echipei React: efectul rezistă unei demontări și remontări rapide? Dacă da, este corect; dacă nu, este nevoie de o funcție de cleanup. [7]

> **Notă**
>
> În producție, `<StrictMode>` nu are niciun efect, iar `useEffect` rulează exact așa cum este scris. Comportamentul de dublare apare *doar în dezvoltare*. Dacă observați în consolă două log-uri „Se încarcă produsele…” la pornire, este Strict Mode în acțiune; nu e o problemă a codului vostru. Pentru cererile HTTP simple, dublarea este inofensivă (la a doua rulare, datele se rescriu peste cele identice).

7. Prima componentă care încarcă date

Să punem împreună `useEffect`, `axios` și `state`. Construim, pas cu pas, o componentă `ProductsList` care încarcă lista de produse de la backend și o afișează.

7.1. Versiunea minimă

```
import { useEffect, useState } from "react";
import axios from "axios";
import ProductCard from "./ProductCard.jsx";

function ProductsList() {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    axios.get("http://localhost:8080/api/products")
      .then(response => setProducts(response.data));
  }, []);

  return (
    <div className="product-list">
      {products.map(p => <ProductCard key={p.id} product={p} />)}
    </div>
  );
}

export default ProductsList;
```

Această versiune funcționează și acoperă cazul fericit (*happy path*), dar este insuficientă din mai multe puncte de vedere. Ce se vede pe ecran cât timp cererea HTTP este pe drum? Lista este goală, deci nu apare nimic – utilizatorul nu știe dacă aplicația este blocată sau dacă pur și simplu nu există produse. Ce se întâmplă dacă serverul este oprit sau întoarce o eroare?

Cererea eșuează, dar utilizatorul nu primește nicio indicație – iar dezvoltatorul nici atât (eroarea ajunge într-o promisiune neprinsă, vizibilă doar în consolă).

7.2. Versiunea completă: cele patru stări

O componentă care comunică cu un server are, în orice moment, una dintre patru stări vizibile pentru utilizator [7, 31]:

- **Încărcare** – cererea HTTP este pe drum; afișăm un indicator (spinner sau text „Se încarcă...”);
- **Eroare** – cererea a eșuat (server picat, rețea, CORS, eroare neașteptată); afișăm un mesaj clar;
- **Listă goală** – cererea a reușit, dar nu există date; afișăm un mesaj informativ („Nu există produse înregistrate”);
- **Listă populată** – cazul „fericit”; afișăm efectiv conținutul.

Toate cele patru cazuri trebuie tratate explicit. Componenta arată, în forma completă, astfel:

```
import { useEffect, useState } from "react";
import axios from "axios";
import ProductCard from "./ProductCard.jsx";

function ProductsList() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading]   = useState(true);
  const [error, setError]       = useState(null);

  useEffect(() => {
    axios.get("http://localhost:8080/api/products")
      .then(response => setProducts(response.data))
      .catch(err => setError(err))
      .finally(() => setLoading(false));
  }, []);

  if (loading) {
    return <p className="status">Se încarcă produsele...</p>;
  }

  if (error) {
    return (
      <p className="status status-error">
        Eroare la încărcarea produselor: {error.message}
      </p>
    );
  }

  if (products.length === 0) {
    return <p className="status">Nu există produse înregistrate.</p>;
  }

  return (
    <div className="product-list">
      {products.map(p => <ProductCard key={p.id} product={p} />)}
```

```
      </div>
    );
  }
  export default ProductsList;
```

7.2.1. Comentarii pe componenta completă

Trei variabile de stare – `products` (lista efectivă), `loading` (boolean, inițial `true` – la primul render, cererea este în curs), `error` (inițial `null`).

Fluxul `useEffect` – pornim cererea, în `.then` setăm produsele, în `.catch` setăm eroarea, în `.finally` (rulează indiferent de succes sau eșec) marcăm sfârșitul încărcării. Acest lanț promise–`.then`–`.catch`–`.finally` este pattern-ul standard.

Întoarcere timpurie (early return) – în loc să încercăm să tratăm toate cele patru cazuri într-un singur JSX cu ternari imbricați, returnăm prematur pentru fiecare caz incomplet. Stilul este idiomatic în React și mult mai citibil.

7.3. Folosirea componentei în App

Componenta `ProductsList` înlocuiește integral logica de afișare a listei din App. Acesta din urmă devine mai simplă:

```
import { useState } from "react";
import ProductsList from "./ProductsList.jsx";
import "./App.css";

function App() {
  const [filtru, setFiltru] = useState("");

  return (
    <div className="app">
      <header>
        <h1>Catalog produse</h1>
        <input
          className="search"
          placeholder="Caută..."
          value={filtru}
          onChange={e => setFiltru(e.target.value)}
        />
      </header>

      <main>
        <ProductsList />
      </main>
    </div>
  );
}

export default App;
```

Filtrarea client-side, prezentă în laboratorul anterior în interiorul `App`, va trebui acum mutată în `ProductsList` (sau, mai elegant, transmisă prin props) – discuția face obiectul unuia dintre exercițiile finale.

8. Greșeli uzuale și cum se evită

8.1. Greșeala 1: lipsa array-ului de dependențe

```
useEffect(() => {
  axios.get("/api/products")
    .then(r => setProducts(r.data));
});  // FĂRĂ [] la final - GREȘIT
```

Fără array, efectul rulează după fiecare render. La fiecare apel HTTP reușit, `setProducts` modifică starea, care declanșează un nou render, care declanșează din nou efectul **ciclu infinit**. Serverul primește mii de cereri pe secundă și browserul se blochează. [7]

Fix: adăugați `[]` ca al doilea argument al `useEffect`, dacă efectul trebuie să ruleze o singură dată.

8.2. Greșeala 2: dependențele lipsă

```
function Detail({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    axios.get(`/api/products/${productId}`)
      .then(r => setProduct(r.data));
  }, []);  // GREȘIT - lipsește productId

  // ...
}
```

Cu [ ] gol, efectul rulează doar la primul render. Dacă `productId` se schimbă (părintele transmite alt id), componenta *nu mai re-încarcă* datele – continuă să afișeze produsul vechi. Pluginul `eslint-plugin-react-hooks` (activ implicit în Vite) detectează această problemă și o subliniază în editor: *"React Hook useEffect has a missing dependency: 'productId'"*.

Fix: includeți toate variabilele folosite în interiorul efectului care vin din afara lui (props, state) în array-ul de dependențe:

```
useEffect(() => {
  axios.get(`/api/products/${productId}`)
    .then(r => setProduct(r.data));
}, [productId]);  // CORECT
```

8.3. Greșeala 3: tratarea greșită a erorilor

```
useEffect(() => {
```

```
    axios.get("/api/products")
      .then(r => setProducts(r.data));
}, []);  // Lipsește .catch - errors silente
```

Dacă serverul este oprit sau cererea eșuează din alt motiv, eroarea este absorbită silențios. Utilizatorul vede ecranul de încărcare pentru totdeauna; dezvoltatorul vede o promisiune neprinsă în consolă, fără explicație în UI.

Fix: tratați explicit ramura de eroare (cu `.catch`) și ramura de finalizare (`.finally`), cum am arătat în varianta completă a ProductsList.

8.4. Greșeala 4: setarea state-ului pe componentă demontată

Există un caz mai subtil: utilizatorul navighează rapid de la o componentă la alta, înainte ca cererea HTTP a primei componente să se încheie. Când cererea revine, componenta deja nu mai există. Apelul `setProducts()` pe o componentă demontată este, în versiuni mai vechi de React, sursă de avertismente; în versiunile recente este tratată tăcut, dar tot reprezintă o ineficiență. Soluția corectă este `AbortController`:

```
useEffect(() => {
  const controller = new AbortController();

  axios.get("/api/products", { signal: controller.signal })
    .then(r => setProducts(r.data))
    .catch(err => {
      if (err.name !== "CanceledError") {
        setError(err);
      }
    })
    .finally(() => setLoading(false));

  return () => controller.abort();  // cleanup
}, []);
```

`AbortController` este un API de browser care permite anularea cererilor în curs. La cleanup (componentă demontată sau dependențe schimbate), apelăm `controller.abort()` – axios respinge cererea cu o eroare specială `CanceledError`, pe care o ignorăm explicit pentru a nu o afișa ca eroare reală. Pentru acest laborator, putem omite această protecție și o adăugăm ca exercițiu opțional; este însă bine să o cunoașteți pentru aplicații reale. [11]

9. Separarea pe straturi: modulul productApi

Componenta `ProductsList` funcționează corect, dar are o problemă de organizare: codul de comunicare cu serverul (URL-ul, apelul `axios`, structura cererii) este amestecat cu logica de afișare. Pentru o aplicație cu mai multe componente care trebuie să acceseze același backend, această amestecare devine rapid problematică [38, 39]:

- URL-ul serverului ar fi duplicat în fiecare componentă;

- schimbarea de la `/api/products` la `/api/v2/products` ar însemna modificarea în multe locuri;
- antetele comune (autentificare, format) ar trebui adăugate la fiecare apel separat;
- testarea componentei devine dificilă (cum simulăm un răspuns `axios` într-un test unitar?).

Soluția este aceeași ca pe backend: separarea pe straturi. Pe frontend, primul strat care merită extras este cel de comunicare cu serverul. Creăm un nou director `src/api/` și în el un fișier `productApi.js`:

```
// src/api/productApi.js

import axios from "axios";

const API_BASE_URL = "http://localhost:8080/api";

const apiClient = axios.create({
  baseURL: API_BASE_URL,
  headers: {
    "Content-Type": "application/json",
  },
  timeout: 10000,
});

export const productApi = {
  getAll: () =>
    apiClient.get("/products").then(r => r.data),

  getById: (id) =>
    apiClient.get(`/products/${id}`).then(r => r.data),
};
```

9.1. Anatomia modulului

`axios.create({...})` creează o *instanță axios* cu configurație implicită. Toate apelurile făcute prin această instanță moștenesc baseURL-ul, antetele și timeout-ul. Configurând o singură dată în acest loc, evităm repetarea în fiecare apel.

`baseURL: API_BASE_URL` – toate apelurile vor avea automat ca prefix `http://localhost:8080/api`. Apelul `apiClient.get("/products")` merge la `http://localhost:8080/api/products`.

`timeout: 10000` – dacă răspunsul nu vine în 10 secunde, axios respinge cererea cu eroare de timeout. Util pentru a nu lăsa utilizatorul să aștepte la nesfârșit dacă serverul este indisponibil.

`productApi` este un obiect care expune metodele relevante. Fiecare metodă întoarce o promisiune cu *datele deja extrase* (am făcut `.then(r => r.data)` în interior). Consumatorii nu mai trebuie să cunoască structura `response` a axios – primesc direct

array-ul de produse sau obiectul produs. Această convenție păstrează codul componentelor curat. [10]

9.2. Componenta refactorizată

Folosind `productApi`, `ProductsList` devine mai curată:

```
import { useEffect, useState } from "react";
import { productApi } from "./api/productApi";
import ProductCard from "./ProductCard.jsx";

function ProductsList() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading]   = useState(true);
  const [error, setError]       = useState(null);

  useEffect(() => {
    productApi.getAll()
      .then(setProducts)
      .catch(setError)
      .finally(() => setLoading(false));
  }, []);

  if (loading) {
    return <p className="status">Se încarcă produsele...</p>;
  }
  if (error) {
    return (
      <p className="status status-error">
        Eroare: {error.message}
      </p>
    );
  }
  if (products.length === 0) {
    return <p className="status">Nu există produse.</p>;
  }

  return (
    <div className="product-list">
      {products.map(p => <ProductCard key={p.id} product={p} />)}
    </div>
  );
}

export default ProductsList;
```

Dacă facem o comparație cu varianta directă:

- nu mai există import axios;
- nu mai există URL-uri scrise în corpul componentei;
- nu mai există acces la response.data.

Componenta cunoaște exclusiv conceptul de „produs”. Schimbarea backend-ului către altă adresă (sau introducerea unui proxy) se face într-un singur loc – în `productApi.js`.

Sintaxa `.then(setProducts)` (în loc de `.then(data => setProducts(data))`) este o scurtătură funcțională validă: când singurul rol al callback-ului este să paseze argumentul către altă funcție cu aceeași semnătură, putem trece direct funcția. Același lucru cu `.catch(setError)`.

10. Variabile de mediu pentru baseURL

Adresa `http://localhost:8080/api` hardcodată în `productApi.js` funcționează în dezvoltare, dar nu va merge în producție. Soluția standard este externalizarea în *variabile de mediu*, iar Vite suportă acest mecanism nativ. [8]

Creăm, la rădăcina proiectului React, un fișier `.env.development`:

```
VITE_API_URL=http://localhost:8080/api
```

Și un fișier `.env.production` (gol pentru moment, va conține URL-ul real al serverului în producție):

```
VITE_API_URL=https://produse.example.com/api
```

Important: prefixul `VITE_` este obligatoriu. Vite expune codului doar variabilele care încep cu acest prefix; restul rămân disponibile doar serverului de build, ca măsură de securitate (pentru a nu expune accidental secrete în bundle-ul final).

În `productApi.js`, folosim variabila prin `import.meta.env`:

```
const API_BASE_URL = import.meta.env.VITE_API_URL;

const apiClient = axios.create({
  baseURL: API_BASE_URL,
  headers: { "Content-Type": "application/json" },
  timeout: 10000,
});
```

Vite alege automat fișierul potrivit: `.env.development` când rulăm `npm run dev`, `.env.production` când rulăm `npm run build`. Există și `.env.local` pentru variabile specifice unei mașini de dezvoltare (ignorat de git, prin convenție). Detaliile complete sunt în documentația oficială Vite.

> **Notă privind git-ul**
>
> Fișierele `.env.development` și `.env.production` sunt versionate (cu valori non-sensibile). Fișierul `.env.local`, precum și orice `.env.*.local`, sunt automat în `.gitignore` (Vite generează această configurație implicit). Acolo se pun valori personale sau sensibile (chei API de testare, etc.) care nu trebuie să ajungă în repository.

11. Aplicația finală a laboratorului

Punem totul cap la cap. Structura proiectului React, după acest laborator:

```
products-ui/
├── .env.development        ← VITE_API_URL pentru dev
├── .env.production         ← VITE_API_URL pentru producție
├── package.json            ← include dependența axios
└── src/
    ├── api/
    │   └── productApi.js   ← strat de comunicare cu backend-ul
    ├── App.css
    ├── App.jsx             ← componenta rădăcină, mai simplă
    ├── ProductCard.jsx     ← neschimbată
    ├── ProductsList.jsx    ← componentă nouă, conectată la API
    ├── StockBadge.jsx      ← neschimbată
    ├── index.css
    └── main.jsx            ← punctul de intrare
```

Pe partea de backend, am adăugat o singură clasă:

```
restlab/
└── src/main/java/ro/unitbv/restlab/
    └── config/
        └── CorsConfig.java   ← clasă nouă, configurare CORS globală
```

11.1. Pași de verificare

Pentru a confirma că totul funcționează, urmăm secvența:

1. **Backend.** Pornim aplicația Spring Boot. Verificăm cu `curl` că antetele CORS sunt prezente.
2. **Frontend.** În alt terminal, în directorul `products-ui`, rulăm `npm run dev`. Vite pornește pe portul 5173.
3. **Browser.** Deschidem `http://localhost:5173`. Pentru o secundă apare „Se încarcă produsele...”, apoi cardurile populate cu date reale din baza de date a backend-ului.
4. **Verificare în DevTools.** Tab-ul Network al browserului arată cererea către `/api/products` cu statusul 200, antetele `Access-Control-*` prezente, răspunsul JSON. Tab-ul React DevTools arată state-ul componentei `ProductsList`: `products: [...], loading: false, error: null`.
5. **Test al fluxului de eroare.** Oprim backend-ul. Reîmprospătăm pagina React. După 10 secunde (timeout-ul axios), apare „Eroare: timeout of 10000ms exceeded”. Repornim backend-ul, reîmprospătăm – funcționează din nou.

11.2. Ce am realizat

Cu acest laborator, aplicația React nu mai este izolată. Lista de produse afișată reflectă starea reală a bazei de date a aplicației Spring Boot. Toată afișarea (componentele `ProductCard` și `StockBadge`) a rămas neschimbată – doar sursa datelor s-a mutat din array-ul hardcodat în apel HTTP. Această invarianță a stratului de prezentare la schimbarea sursei datelor este exact ce promite arhitectura React de la început, și ce face acum vizibil.

12. Exerciții propuse

12.1. Exercițiul 1 – Buton de reîncărcare manuală

Modificați componenta `ProductsList` astfel încât logica de încărcare să fie extrasă într-o funcție `loadProducts()` definită în interiorul componentei (înaintea `useEffect`). Această funcție este apelată atât din `useEffect`, cât și de la click pe un buton „Reîncarcă lista" adăugat undeva vizibil.

Atenție: la fiecare apel manual al `loadProducts`, trebuie resetate explicit stările – `loading` la `true`, `error` la `null` – altfel UI-ul nu va arăta corect tranziția între o încercare anterioară și cea nouă.

12.2. Exercițiul 2 – Filtrarea client-side

Re-introduceți funcționalitatea de filtrare după nume care exista în laboratorul anterior. Decizii de luat: starea `filtru` rezidă în `App` sau în `ProductsList`? Dacă rămâne în `App`, cum o transmite către `ProductsList`? Justificați alegerea.

Indicație: dacă o transmiteți prin `props`, `ProductsList` o primește ca `props.filtru` și o aplică pe `products` înainte de `.map`. Filtrarea este pur client-side – nu produce o nouă cerere HTTP la fiecare tastă apăsată.

12.3. Exercițiul 3 – Pagina de detaliu pentru un produs

Construiți o componentă `ProductDetail` care primește un id ca prop, apelează `productApi.getById(id)` și afișează detaliile unui produs (nume, preț, stoc), cu tratarea celor patru stări (loading, error, not found, populated).

Tratați explicit cazul 404 (produs inexistent). În `axios`, eroarea ajunge pe ramura `.catch` cu `error.response.status === 404`. Distingeți acest caz de erorile reale (server deconectat, etc.) – afișați un mesaj specific „Produsul cu id-ul X nu există" diferit de „Eroare la încărcarea produsului".

Pentru a testa componenta, modificați `App` să țină în state un id selectat și, când există, să afișeze `<ProductDetail id={selectatId} />` în panoul lateral.

12.4. Exercițiul 4 – Externalizarea baseURL în .env

Implementați mecanismul de variabile de mediu descris în acest laborator. Creați `.env.development` și `.env.production`, modificați `productApi.js` să citească `import.meta.env.VITE_API_URL`.

Verificați că aplicația funcționează identic. Rulați apoi `npm run build` urmat de `npm run preview` – build-ul de producție va folosi adresa din `.env.production`. Confirmați (cu DevTools → Network) că adresa folosită este cea așteptată.

12.5. Exercițiul 5 – Anularea cererilor

Implementați protecția cu `AbortController` în `ProductsList`, descrisă în secțiunea de greșeli. Apoi simulați comportamentul: introduceți artificial un `setTimeout` în interiorul controllerului Spring Boot (`Thread.sleep(5000)`) astfel încât cererea să dureze 5 secunde. În frontend, navigați rapid (deschideți / închideți componenta de mai multe ori). Verificați în Network că cererile vechi sunt marcate ca anulate.

Laborator 9: Operații CRUD complete. Formulare controlate și tratarea erorilor de validare

1. Obiectivul laboratorului

Până în acest moment, aplicația React realizată poate doar să citească date de pe backend. În acest laborator vom completa cele patru operații ale modelului CRUD: *Create* (adăugare de produs nou), *Read* (deja implementat), *Update* (modificarea unui produs existent) și *Delete* (ștergerea). La final, utilizatorul va putea gestiona integral catalogul de produse prin interfața React, fără a mai avea nevoie de Postman sau de alte instrumente externe.

Tema centrală a laboratorului este conceptul de formular controlat – o tehnică prin care valoarea fiecărui câmp dintr-un formular este sincronizată, în orice moment, cu starea componentei React. Pe această fundație se va construi formularul de creare, se va reutiliza pentru editare, se vor consuma erorile de validare returnate de Spring Boot și se vor afișa elegant lângă câmpurile corespunzătoare. Ștergerea va fi protejată printr-o confirmare, iar fiecare operație va oferi feedback vizibil utilizatorului prin notificări de tip toast.

La finalul laboratorului, cititorul ar trebui să fie capabil:

- să explice conceptul de input controlat și să motiveze de ce React preferă această abordare;
- să construiască formulare cu mai multe câmpuri, gestionate prin `useState` (state per câmp sau state-obiect);
- să trimită date către server cu `axios` prin metodele `POST`, `PUT` și `DELETE`;
- să distingă validarea client-side de validarea server-side și să motiveze de ce ambele sunt necesare;
- să consume corect răspunsul `ValidationExceptionResponse` din Spring Boot și să afișeze erorile pe câmpurile corespunzătoare;
- să implementeze ștergerea cu confirmare și actualizarea optimistă a listei;
- să ofere utilizatorului feedback vizibil (succes, eroare) printr-o componentă de notificare reutilizabilă;
- să identifice și să elimine duplicarea dintre formularele de creare și de editare prin refactorizare.

2. Formulare controlate în React

Într-un formular HTML clasic, valoarea unui input este ținută în DOM-ul browserului. Codul JavaScript o citește atunci când are nevoie (de obicei, la submit) prin

`document.getElementById` sau alte API-uri DOM. Acest model se numește, în terminologia React, *input necontrolat* (uncontrolled): valoarea există în DOM, separat de starea componentei. [7, 16]

Într-un formular controlat, valoarea fiecărui input este *stocată în starea componentei*, injectată în input prin atributul `value`, iar orice modificare în input declanșează un eveniment `onChange` care actualizează starea. Astfel, starea componentei este singura sursă de adevăr pentru valorile din formular.

Cele două abordări au consecințe practice diferite. Tabelul următor le sintetizează.

Aspect	***Input necontrolat***	***Input controlat***
Sursa de adevăr	DOM-ul browserului	Starea componentei
Citirea valorii	Prin ref sau la submit	Direct, oricând, din state
Validare în timp real	Dificilă	Naturală
Manipulare programatică (resetare, autocompletare)	Necesită apeluri DOM	Simplă setare a state-ului
Cod scris	Mai puțin	Mai mult (un onChange per câmp)

În cadrul acestui îndrumar folosim exclusiv formulare controlate. Volumul de cod suplimentar este modest pentru câmpuri puține, iar avantajele – validare directă, transformări programatice ușoare, integrare naturală cu logica de business – sunt importante.

2.1. Anatomia unui input controlat

Cel mai simplu exemplu este un input cu un singur câmp:

```
import { useState } from "react";

function ExempluInput() {
  const [nume, setNume] = useState("");

  return (
    <div>
      <input
        type="text"
        value={nume}
        onChange={e => setNume(e.target.value)}
      />
      <p>Ai scris: {nume}</p>
    </div>
  );
}
```

Trei elemente sunt esențiale: `value={nume}` injectează valoarea curentă a stării în input – la fiecare re-randare, input-ul afișează exact ce e în `nume`; `onChange={e => setNume(e.target.value)}` actualizează starea la fiecare modificare, citind valoarea brută din `e.target.value`; iar inițializarea `useState("")` trebuie să fie un șir gol (nu `undefined`), altfel React produce o avertizare despre comutarea între un input necontrolat și unul controlat. [7]

2.2. Două strategii pentru formulare cu mai multe câmpuri

Aplicația noastră are nevoie de un formular cu trei câmpuri: nume, preț, stoc. Există două strategii idiomatice pentru a gestiona această stare. [7, 30]

2.2.1. Strategia 1: un useState per câmp

```
const [name, setName]   = useState("");
const [price, setPrice] = useState("");
const [stock, setStock] = useState("");
```

Avantaje: cod explicit, ușor de citit, fiecare câmp are setter-ul lui dedicat.

Dezavantaj: pentru formulare mari (10+ câmpuri), repetiția devine obositoare.

2.2.2. Strategia 2: un singur useState cu obiect

```
const [form, setForm] = useState({
  name: "",
  price: "",
  stock: "",
});

const handleChange = (e) => {
  setForm({ ...form, [e.target.name]: e.target.value });
};

// În JSX, fiecare input primește atributul "name":
<input name="name"  value={form.name}  onChange={handleChange} />
<input name="price" value={form.price} onChange={handleChange} />
<input name="stock" value={form.stock} onChange={handleChange} />
```

Avantaje: un singur handler pentru toate câmpurile, structură compactă, ușor de extins (un câmp nou înseamnă un singur nou rând în JSX).

Dezavantaj: sintaxa `[e.target.name]: ...` (proprietate calculată) este mai puțin imediată pentru începători.

> **Atenție**
>
> Apelul `setForm({ ...form, name: "X" })` este obligatoriu să facă spread. Dacă scriem `setForm({ name: "X" })`, state-ul devine `{ name: "X" }` – celelalte câmpuri (price, stock) dispar. `useState` *nu îmbină* obiectul nou cu cel vechi, ca `setState` din componente clasă; îl înlocuiește integral.

În cadrul acestui laborator folosim strategia 1 (un `useState` per câmp) pentru claritate didactică. Strategia 2 face obiectul unuia dintre exercițiile finale.

3. Formularul de creare a unui produs

Se va construi gradual componenta `CreateProductForm`. Se pornește de la o variantă minimă, care doar afișează inputurile și înregistrează valorile, apoi se va adăuga submit-ul, apoi tratarea erorilor de validare. [7, 31]

3.1. Versiunea 1: JSX și state

```
import { useState } from "react";

function CreateProductForm() {
  const [name, setName]   = useState("");
  const [price, setPrice] = useState("");
  const [stock, setStock] = useState("");

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log({ name, price, stock });
  };

  return (
    <form onSubmit={handleSubmit} className="product-form">
      <div className="field">
        <label htmlFor="name">Nume produs</label>
        <input
          id="name"
          type="text"
          value={name}
          onChange={e => setName(e.target.value)}
        />
      </div>

      <div className="field">
        <label htmlFor="price">Preț (lei)</label>
        <input
          id="price"
          type="number"
          step="0.01"
          value={price}
          onChange={e => setPrice(e.target.value)}
        />
      </div>

      <div className="field">
        <label htmlFor="stock">Stoc</label>
        <input
          id="stock"
          type="number"
          value={stock}
          onChange={e => setStock(e.target.value)}
        />
      </div>

      <button type="submit">Salvează</button>
```

```
      </form>
    );
  }

  export default CreateProductForm;
```

> **Observații**
>
> `e.preventDefault()` este obligatoriu în handler-ul de submit, altfel browserul încearcă să trimită formularul prin GET către URL-ul curent, ceea ce înseamnă ieșirea din SPA și pierderea stării. Atributul `htmlFor` pe `<label>` corespunde lui `for` din HTML (cuvânt rezervat în JavaScript); el leagă eticheta de input prin atributul `id`, făcând astfel ca un click pe etichetă să transfere focusul pe input – important pentru accesibilitate.
>
> Tipurile `type="number"` activează tastatura numerică pe dispozitive mobile și restricționează (parțial) ce se poate scrie. Important: valoarea citită din `e.target.value` rămâne un șir de caractere, chiar și pentru `type="number"`. Conversia la număr o vom face la momentul submit-ului, când trimitem datele către backend.

3.2. Versiunea 2: submit cu axios

Se modifică handler-ul de submit pentru a apela `productApi.create` (metoda nouă în modulul `productApi`). Mai întâi, extinderea modulului:

```
// src/api/productApi.js (extins)
export const productApi = {
  getAll: () =>
    apiClient.get("/products").then(r => r.data),

  getById: (id) =>
    apiClient.get(`/products/${id}`).then(r => r.data),

  create: (data) =>
    apiClient.post("/products", data).then(r => r.data),

  update: (id, data) =>
    apiClient.put(`/products/${id}`, data).then(r => r.data),

  remove: (id) =>
    apiClient.delete(`/products/${id}`),
};
```

Se introduc trei metode noi. Convențional, metoda DELETE a fost numită `remove` în loc de `delete` pentru că `delete` este cuvânt rezervat în JavaScript și ar produce erori dacă s-ar folosi ca nume de funcție sau proprietate. [10]

Acum, componenta cu submit real:

```
import { useState } from "react";
```

```
import { productApi } from "./api/productApi";

function CreateProductForm({ onSuccess }) {
  const [name, setName]   = useState("");
  const [price, setPrice] = useState("");
  const [stock, setStock] = useState("");

  const handleSubmit = async (e) => {
    e.preventDefault();

    const newProduct = await productApi.create({
      name: name,
      price: Number(price),
      stock: Number(stock),
    });

    // Resetăm formularul
    setName("");
    setPrice("");
    setStock("");

    // Anunțăm componenta părinte
    onSuccess(newProduct);
  };

  return (
    /* JSX neschimbat */
  );
}
```

Observații

`Number(price)` convertește șirul din state într-un număr; este obligatoriu, altfel backend-ul ar primi `{price: "199.90"}` (șir) în loc de `{price: 199.90}` (număr), iar Jackson ar respinge cererea cu eroare de deserializare.

Prop-ul `onSuccess` este un callback transmis de părinte. După ce produsul este salvat cu succes, anunțăm părintele – care, de regulă, va adăuga produsul nou în lista afișată sau va reîncărca lista. Aceasta este aplicarea principiului „date în jos, evenimente în sus”.

3.3. Versiunea 3: starea de submit și gestionarea erorilor

Versiunea 2 are două probleme. Prima: dacă utilizatorul apasă rapid butonul de două ori (sau ține apăsat din greșeală), aplicația va trimite două cereri POST, creând două produse duplicate. A doua: dacă cererea eșuează (validare server-side, server picat, eroare de rețea), nu se primește nicio indicație în UI. Versiunea 3 le tratează pe ambele. [7]

```
function CreateProductForm({ onSuccess }) {
  const [name, setName]   = useState("");
```

```
  const [price, setPrice] = useState("");
  const [stock, setStock] = useState("");

  const [submitting, setSubmitting] = useState(false);
  const [errors, setErrors] = useState({});

  const handleSubmit = async (e) => {
    e.preventDefault();
    setSubmitting(true);
    setErrors({});

    try {
      const newProduct = await productApi.create({
        name: name,
        price: Number(price),
        stock: Number(stock),
      });
      setName(""); setPrice(""); setStock("");
      onSuccess(newProduct);
    } catch (err) {
      const status = err.response?.status;
      const fieldErrors = err.response?.data?.validationErrors;

      if (status === 400 && fieldErrors) {
        setErrors(fieldErrors);
      } else {
        setErrors({ _global: "A apărut o eroare la salvare." });
      }
    } finally {
      setSubmitting(false);
    }
  };

  // ... JSX, cu submit dezactivat dacă submitting === true
}
```

Două variabile de stare noi:

- **`submitting`** – un boolean care marchează intervalul dintre momentul în care utilizatorul apasă submit și momentul în care primim răspuns. Cât timp e `true`, butonul este dezactivat în JSX.
- **`errors`** – un obiect care colectează mesajele de eroare per câmp, plus un câmp special `_global` pentru erori care nu țin de un câmp anume. Inițial este `{}` (gol).

Fluxul `try / catch / finally` este structura standard pentru o operație asincronă cu cele trei stări: succes, eșec, finalizare (rulează în ambele cazuri). În `finally`, se va inițializa `submitting` la `false` – butonul redevine activ.

3.4. JSX-ul actualizat: erori și buton dezactivat

```
  return (
    <form onSubmit={handleSubmit} className="product-form">
```

```
      {errors._global && (
        <div className="alert alert-error">{errors._global}</div>
      )}

      <div className="field">
        <label htmlFor="name">Nume produs</label>
        <input
          id="name"
          type="text"
          value={name}
          onChange={e => setName(e.target.value)}
          disabled={submitting}
        />
        {errors.name && (
          <span className="field-error">{errors.name}</span>
        )}
      </div>

      <div className="field">
        <label htmlFor="price">Preț (lei)</label>
        <input
          id="price" type="number" step="0.01"
          value={price}
          onChange={e => setPrice(e.target.value)}
          disabled={submitting}
        />
        {errors.price && (
          <span className="field-error">{errors.price}</span>
        )}
      </div>

      <div className="field">
        <label htmlFor="stock">Stoc</label>
        <input
          id="stock" type="number"
          value={stock}
          onChange={e => setStock(e.target.value)}
          disabled={submitting}
        />
        {errors.stock && (
          <span className="field-error">{errors.stock}</span>
        )}
      </div>

      <button type="submit" disabled={submitting}>
        {submitting ? "Se salvează..." : "Salvează"}
      </button>
    </form>
  );
```

Sunt trei pattern-uri vizibile aici:

- Eroarea per câmp – pentru fiecare câmp, se afișează sub input `{errors.name && <span ...>}`. Dacă `errors.name` este `undefined` (cazul fericit), nu se randează nimic; dacă conține un mesaj, apare în elementul `span`.
- Eroarea globală – la începutul formularului, `{errors._global && ...}` afișează un mesaj de tip „bandă roșie” pentru eșecuri care nu țin de un câmp anume (server picat, eroare neclară).
- Dezactivarea cu `disabled={submitting}` – pe toate inputurile și pe buton. Vizibil, butonul își schimbă și textul în „Se salvează...”, oferind feedback clar utilizatorului.

4. Validarea pe două niveluri

Validarea formularelor este o problemă mai subtilă într-o aplicație SPA decât în aplicațiile clasice. Există două niveluri care nu se exclud, ci se completează:

4.1. Validarea server-side

Validarea server-side este realizată de Spring Boot prin Bean Validation (deja implementată). Este *obligatorie pentru integritatea datelor*. Indiferent ce face frontend-ul, serverul trebuie să-și protejeze baza de date de date invalide. Un atacator poate ocoli oricând JavaScript-ul – poate folosi Postman, `curl`, sau un script propriu – și să trimită direct cereri către API. Singura barieră reală este la nivelul serverului. [7, 20]

În aplicația `restlab`, validarea este declarată pe `CreateProductRequest` prin adnotări (`@NotBlank`, `@Positive`, `@PositiveOrZero` etc.), iar `GlobalExceptionHandler` transformă `MethodArgumentNotValidException` în corpul de tip `ValidationExceptionResponse` trimis cu cod 400.

4.2. Validarea client-side

Validarea client-side este realizată în browser, înainte ca cererea să fie trimisă către server. Rolul ei nu este de securitate (ar fi ineficient), ci de *experiență a utilizatorului*: feedback imediat (fără round-trip de 100-500 ms), reducerea traficului inutil, posibilitatea de a sublinia exact câmpul problematic înainte ca utilizatorul să apese submit. [7]

Strategia folosită este: validarea client-side este o îmbunătățire opțională, aplicată peste validarea server-side. Se pornește de la zero validare client-side – formularul trimite cererea, iar dacă e invalidă, primește 400 și afișează erorile venite de la server. Apoi, dacă se dorește, se adăugă preventiv validări locale care să prindă greșelile evidente înainte de submit.

4.2.1. Formatul ValidationExceptionResponse

Răspunsul Spring Boot pentru o validare eșuată arată astfel:

```
{
  "timestamp": "2026-05-16T15:30:42",
  "status": 400,
  "error": "Bad Request",
```

```
  "validationErrors": {
    "name":  "Numele produsului este obligatoriu.",
    "price": "Pretul trebuie sa fie strict pozitiv."
  },
  "path": "/api/products"
}
```

Câmpul cheie pentru frontend este `validationErrors` – un obiect (`Map<String, String>` pe backend) de la numele câmpului la mesajul de eroare. Forma este aproape identică cu obiectul `errors` din componenta React, ceea ce face maparea trivială:

```
} catch (err) {
  const fieldErrors = err.response?.data?.validationErrors;
  if (err.response?.status === 400 && fieldErrors) {
    setErrors(fieldErrors);
  }
}
```

Cu o singură instrucțiune – `setErrors(fieldErrors)` – mesajele de pe server ajung lângă câmpurile React. Numele câmpurilor din DTO-ul Spring Boot trebuie să coincidă cu numele variabilelor de stare din React, lucru care este natural dacă sunt urmate convențiile de denumire în ambele părți. Această *punte directă* între backend și frontend este o consecință a unei alegeri arhitecturale bune – DTO-uri cu nume descriptive – iar răsplata o vedem aici, sub forma simplității integrării.

> **Notă privind operatorul ?**
>
> Notația `err.response?.data?.validationErrors` folosește *optional chaining* (ECMAScript 2020). Dacă `err.response` este `undefined` (de exemplu, cazul în care cererea nici nu a ajuns la server), întreaga expresie este `undefined` în loc să arunce o eroare `TypeError`. Operatorul este indispensabil pentru navigarea în structuri care pot fi parțial absente.

4.2.2. Validare client-side preventivă

Se poate adăuga, înainte de a apela `axios`, o funcție de validare locală care să prindă cele mai evidente greșeli – câmpuri goale, valori non-numerice, prețuri negative – fără a face cerere către server. Cheile obiectului produs sunt aceleași ca cele folosite de validarea server-side, astfel încât UI-ul afișează erorile în același loc, indiferent de sursă.

```
function validateClient({ name, price, stock }) {
  const errs = {};
  if (!name.trim()) {
    errs.name = "Numele este obligatoriu.";
  }
  if (!price || Number(price) <= 0) {
    errs.price = "Prețul trebuie să fie strict pozitiv.";
  }
  if (stock === "" || Number(stock) < 0) {
```

```
    errs.stock = "Stocul trebuie să fie un număr ≥ 0.";
  }
  return errs;
}

// În handleSubmit, înainte de axios:
const clientErrors = validateClient({ name, price, stock });
if (Object.keys(clientErrors).length > 0) {
  setErrors(clientErrors);
  setSubmitting(false);
  return;
}
```

Validarea client-side nu înlocuiește validarea server-side; o complementează. Dacă cele două sunt sincronizate (regulile identice), erorile vor apărea identic; dacă există discrepanțe (regulile server-side sunt mai stricte), erorile server-side au prioritate.

5. Integrarea formularului cu lista de produse

Componenta `CreateProductForm` trăiește acum izolat. Pentru a face parte din aplicație, se poate integra în `ProductsList` (sau în `App`, după preferință), unde adăugarea unui produs nou trebuie să se reflecte imediat în lista afișată. [7, 30]

5.1. Decizie: unde trăiește formularul

Avem două opțiuni de design:

- Opțiunea A: formularul este mereu vizibil, deasupra listei. Simplu, dar consumă spațiu chiar și când utilizatorul doar consultă lista.
- Opțiunea B: există un buton „+ Adaugă produs”, care comută între modul listă și modul formular. Mai compact, dar necesită o stare suplimentară pentru modul curent.

Alegem opțiunea B, care este mai realistă pentru o aplicație de producție. State-ul „mod” rezidă în `App`.

```
function App() {
  const [mode, setMode] = useState("list");
  // mode poate fi: "list" sau "create"

  const handleCreateSuccess = () => {
    setMode("list");
    // ProductsList va re-încărca automat?
    // NU, dacă nu îi spunem să facă asta.
  };

  return (
    <div className="app">
      <header>
        <h1>Catalog produse</h1>
        {mode === "list" && (
          <button onClick={() => setMode("create")}>
```

```
              + Adaugă produs
            </button>
          )}
        </header>

        <main>
          {mode === "list" && <ProductsList />}
          {mode === "create" && (
            <CreateProductForm
              onSuccess={handleCreateSuccess}
              onCancel={() => setMode("list")}
            />
          )}
        </main>
      </div>
    );
  }
```

5.2. Problema: lista nu se actualizează după creare

Comentariul din cod identifică problema: după ce creăm un produs nou și revenim la listă, `ProductsList` afișează datele încărcate la prima sa montare. Cererea `GET /api/products` nu se face din nou. [7]

Există mai multe soluții. Cea mai simplă, dar suficientă pentru moment, este să folosim prop-ul `key` ca mecanism de re-montare:

```
function App() {
  const [mode, setMode] = useState("list");
  const [listKey, setListKey] = useState(0);

  const handleCreateSuccess = () => {
    setListKey(k => k + 1); // forțează re-montarea ProductsList
    setMode("list");
  };

  return (
    /* ... ProductsList primește key={listKey} ... */
    <ProductsList key={listKey} />
  );
}
```

Schimbarea prop-ului `key` face React să trateze componenta ca pe una nouă: o demontează pe cea veche, o montează pe cea nouă. `useEffect(..., [])` rulează din nou și încarcă lista actualizată. Soluția este puțin „forțată” dar simplă; o variantă mai elegantă (transmitere directă a produsului nou către listă) face obiectul unui exercițiu.

6. Formularul de editare

Formularul de editare reia integral structura formularului de creare. Singurele diferențe sunt [30]:

- la montare, valorile inițiale ale state-ului provin de pe server (printr-un GET pentru produsul ce urmează a fi editat);
- submit-ul apelează `productApi.update` în loc de `productApi.create`;
- textul butonului este „Modifică” în loc de „Salvează”.

6.1. Componenta EditProductForm

```
import { useState, useEffect } from "react";
import { productApi } from "./api/productApi";

function EditProductForm({ id, onSuccess, onCancel }) {
  const [name, setName]     = useState("");
  const [price, setPrice]   = useState("");
  const [stock, setStock]   = useState("");
  const [loading, setLoading]     = useState(true);
  const [submitting, setSubmitting] = useState(false);
  const [errors, setErrors] = useState({});

  // Încărcăm datele inițiale ale produsului
  useEffect(() => {
    productApi.getById(id)
      .then(product => {
        setName(product.name);
        setPrice(String(product.price));
        setStock(String(product.stock));
      })
      .finally(() => setLoading(false));
  }, [id]);

  const handleSubmit = async (e) => {
    e.preventDefault();
    setSubmitting(true);
    setErrors({});

    try {
      const updated = await productApi.update(id, {
        name: name,
        price: Number(price),
        stock: Number(stock),
      });
      onSuccess(updated);
    } catch (err) {
      const status = err.response?.status;
      const fieldErrors = err.response?.data?.validationErrors;
      if (status === 400 && fieldErrors) {
        setErrors(fieldErrors);
      } else {
        setErrors({ _global: "A apărut o eroare la salvare." });
      }
    } finally {
      setSubmitting(false);
    }
  };
```

```
  if (loading) return <p>Se încarcă...</p>;

  return (
    /* JSX practic identic cu CreateProductForm */
  );
}

export default EditProductForm;
```

Observații

`useEffect` cu `[id]` – array-ul de dependințe conține `id`, astfel încât dacă părintele transmite alt produs de editat, efectul rulează din nou și încarcă noile date.

Conversiile `String(product.price)` sunt necesare pentru că state-ul `price` ține un șir (input-urile lucrează cu șiruri). Dacă s-ar folosi direct numărul, prima randare ar avea valoarea `{ value: 199.9 }`, care funcționează, dar React va comuta apoi spre șir la prima editare, producând inconsistențe subtile. Convenția este: *state-ul unui input controlat e mereu de tip șir*; conversia se face doar la momentul submit-ului.

6.2. Duplicare evidentă – semnal pentru refactorizare

Comparând `CreateProductForm` și `EditProductForm`, se observă o suprapunere masivă: aceleași state-uri, aceleași inputuri, aceeași tratare a erorilor, aceeași dezactivare la submit. Singurele diferențe sunt cele punctate mai sus. Aceasta este o duplicare evidentă, simptom al unei abstracții ratate. [32, 39]

Refactorizarea naturală este o componentă `ProductForm` parametrizată prin `props`:

```
 <ProductForm
  initialValues={{ name: "", price: "", stock: "" }}
  onSubmit={(data) => productApi.create(data)}
  submitLabel="Salvează"
  onSuccess={handleCreateSuccess}
  onCancel={handleCancel}
/>

<ProductForm
  initialValues={existingProduct}
  onSubmit={(data) => productApi.update(id, data)}
  submitLabel="Modifică"
  onSuccess={handleEditSuccess}
  onCancel={handleCancel}
/>
```

Această refactorizare elimină duplicarea aproape integral și face componenta `ProductForm` reutilizabilă pentru orice scenariu viitor (chiar și pentru clonare – „creează un produs nou pre-populat din cel existent”). Implementarea ei concretă este unul dintre exercițiile finale; pentru moment, păstrăm cele două componente separate, pentru claritate didactică.

7. Ștergerea unui produs

Operația `DELETE` este, din punct de vedere tehnic, cea mai simplă: un apel HTTP, un statusul `204 (No Content)` din partea serverului, fără corp de răspuns. Complexitatea ține de aspectele de experiență a utilizatorului: confirmarea, actualizarea listei, gestionarea eventualelor erori. [18]

7.1. Confirmarea cu window.confirm

Cea mai simplă formă de confirmare este `window.confirm`, un dialog nativ al browserului care blochează firul de execuție până când utilizatorul răspunde:

```
const handleDelete = async (product) => {
  const message = `Sigur doriți să ștergeți produsul "${product.name}"?`;
  const confirmed = window.confirm(message);
  if (!confirmed) return;

  try {
    await productApi.remove(product.id);
    setProducts(prev => prev.filter(p => p.id !== product.id));
  } catch (err) {
    alert("A apărut o eroare la ștergere.");
  }
};
```

Observații

`window.confirm` este sincronă – execuția se oprește efectiv până când utilizatorul apasă un buton. Acesta este unul dintre puținele puncte în care se apelează un API DOM direct, fără a respecta natura asincronă a celorlalte operații.

Dialogul nativ are aspect minimal, nu este personalizabil din punct de vedere vizual și diferă între browsere. Pentru aplicații cu cerințe de UX mai stricte, se construiește un modal React propriu.

7.2. Actualizarea optimistă a listei

După ștergerea cu succes, avem două opțiuni pentru a actualiza UI-ul:

- Pesimist – reîncărcăm toată lista de pe server (GET `/api/products`). Sigur dar lent (un round-trip suplimentar).
- Optimist – eliminăm direct din state elementul șters, fără să așteptăm confirmarea de la server. Mai rapid; bazat pe presupunerea că operația a reușit.

În exemplul de mai sus s-a ales varianta optimistă, prin:

```
setProducts(prev => prev.filter(p => p.id !== product.id));
```

Se observă forma `setProducts(prev => ...)` – varianta funcțională a setter-ului; se folosește aici pentru că noul array depinde de cel anterior; este forma sigură în prezența re-randărilor sau a actualizărilor concurente. `.filter` produce un array nou, fără elementul șters; React detectează schimbarea referinței și re-randează. [33]

8. Feedback prin notificări (toast)

Operațiile CRUD trebuie să ofere feedback vizibil utilizatorului. Până acum s-a folosit `alert()`, care este disruptiv și nu se potrivește unei aplicații moderne.

Soluția standard este o notificare pasageră, numită toast: un mic dreptunghi colorat care apare într-un colț al ecranului, rămâne câteva secunde și dispare automat. În spiritul didactic al îndrumarului, se va construi această componentă manual, fără a recurge la o bibliotecă externă. [33]

8.1. Componenta Toast

```
// src/Toast.jsx
import { useEffect } from "react";

function Toast({ message, type, onClose }) {
  useEffect(() => {
    const timer = setTimeout(onClose, 3000);
    return () => clearTimeout(timer);
  }, [onClose]);

  const className = `toast toast-${type}`;
  return (
    <div className={className} role="alert">
      {message}
      <button className="toast-close" onClick={onClose}>×</button>
    </div>
  );
}

export default Toast;
```

Componenta primește trei props: `message` (textul afișat), `type` ("success" sau "error", determină culoarea), `onClose` (callback care va fi apelat când utilizatorul apasă × sau când expiră timer-ul de 3 secunde).

Punctul interesant al componentei este *funcția de cleanup* a `useEffect`: `() => clearTimeout(timer)`. Este crucială. Dacă utilizatorul închide manual notificarea (sau dacă componenta este demontată din alt motiv) *înainte* ca timer-ul să se declanșeze, cleanup-ul anulează timer-ul. Fără el, `onClose` ar fi apelat pe o componentă deja inexistentă.

Atributul `role="alert"` este adăugată pentru accesibilitate: cititoarele de ecran vor anunța automat conținutul, fără a fi nevoie ca utilizatorul să navigheze către el.

8.2. Folosirea componentei Toast

În componenta părinte (`App` sau `ProductsList`), se adaugă o variabilă de stare `toast` (`null` sau `{message, type}`) și se afișează condițional:

```
function App() {
  const [toast, setToast] = useState(null);

  const showSuccess = (msg) => setToast({ message: msg, type: "success" });
  const showError   = (msg) => setToast({ message: msg, type: "error"   });

  const handleCreateSuccess = (newProduct) => {
    showSuccess(`Produsul "${newProduct.name}" a fost adăugat.`);
    setMode("list");
  };

  return (
    <div className="app">
      {/* ... restul aplicației ... */}

      {toast && (
        <Toast
          message={toast.message}
          type={toast.type}
          onClose={() => setToast(null)}
        />
      )}
    </div>
  );
}
```

Cele două funcții helper `showSuccess` și `showError` sunt transmise prin props către componentele copil care produc evenimente (`CreateProductForm`, `EditProductForm`, butoanele de ștergere). Acolo unde apare o operație CRUD reușită sau eșuată, se apelează una dintre ele.

8.2.1. Stilurile pentru toast

Se adăugă în `App.css`:

```
.toast {
  position: fixed;
  bottom: 24px;
  right: 24px;
  padding: 12px 16px;
  border-radius: 6px;
  box-shadow: 0 4px 12px rgba(0, 0, 0, 0.15);
  display: flex;
  align-items: center;
  gap: 12px;
  z-index: 1000;
  animation: slideIn 0.3s ease-out;
}
```

```
.toast-success { background: #c6f6d5; color: #22543d; }
.toast-error   { background: #fed7d7; color: #742a2a; }

.toast-close {
  background: transparent;
  border: none;
  font-size: 20px;
  cursor: pointer;
  color: inherit;
}

@keyframes slideIn {
  from { transform: translateX(100%); opacity: 0; }
  to   { transform: translateX(0);    opacity: 1; }
}
```

Observații

`position: fixed` scoate toast-ul din fluxul normal al paginii și îl ancorează la viewport (rămâne vizibil la scroll).

`z-index: 1000` îl plasează deasupra restului conținutului.

`@keyframes slideIn` definește o animație simplă de intrare – nu este obligatorie, dar adaugă un detaliu de finețe care face aplicația să pară mai îngrijită.

9. Aplicația finală a laboratorului

Structura proiectului React, după acest laborator:

```
src/
├── api/
│   └── productApi.js      ← extins cu create, update, remove
├── App.css                ← stiluri pentru toast și formular
├── App.jsx          ← orchestrator: state pentru mod, produs editat, toast
├── CreateProductForm.jsx ← componentă nouă
├── EditProductForm.jsx    ← componentă nouă
├── ProductCard.jsx        ← extins cu butoane Editează/Șterge
├── ProductsList.jsx       ← primește acum callback de ștergere
├── StockBadge.jsx         ← neschimbată
├── Toast.jsx              ← componentă nouă
├── index.css
└── main.jsx
```

9.1. Schema interacțiunilor

Pentru orientare, fluxurile sunt:

Creare: utilizatorul apasă „+ Adaugă produs” → `App` comută la mode="create" → randează `CreateProductForm` → submit → `POST /api/products` → la succes, toast verde și revenire la mod="list" cu re-încărcare a `ProductsList`.

Editare: utilizatorul apasă „Editează” pe un card → `App` memorează id-ul și comută la mode="edit" → randează `EditProductForm` cu id-ul respectiv → submit → `PUT /api/products/{id}` → la succes, toast verde și revenire la mod="list".

Ștergere: utilizatorul apasă „Șterge” pe un card → `window.confirm` → la confirmare, `DELETE /api/products/{id}` → la succes, actualizare optimistă a state-ului din `ProductsList` și toast verde.

9.2. ProductCard extins cu butoane

Pentru a permite editarea și ștergerea, `ProductCard` primește două callback-uri suplimentare prin props:

```
function ProductCard({ product, onEdit, onDelete }) {
  return (
    <div className="product-card">
      <h3>{product.name}</h3>
      <div className="product-price">
        {product.price.toFixed(2)} lei
      </div>
      <div className="product-stock">
        Stoc: {product.stock} <StockBadge stock={product.stock} />
      </div>
      <div className="product-actions">
        <button onClick={() => onEdit(product)}>Editează</button>
        <button
          className="btn-danger"
          onClick={() => onDelete(product)}
        >
          Șterge
        </button>
      </div>
    </div>
  );
}
```

10. Exerciții propuse

10.1. Exercițiul 1 – Refactorizare ProductForm

Extrageți logica comună a formularelor de creare și editare într-o singură componentă `ProductForm` care primește prin props: `initialValues` (obiect cu name, price, stock), `onSubmit` (funcție asincronă care primește datele și returnează o promisiune cu rezultatul), `submitLabel` (textul butonului), `onSuccess` și `onCancel` (callback-uri pentru părinte). `CreateProductForm` și `EditProductForm` vor deveni „wrappere” care apelează `ProductForm` cu props corespunzătoare.

10.2. Exercițiul 2 – Modal de confirmare propriu

Înlocuiți `window.confirm` cu un modal React propriu, controlat prin state. Modalul primește prin props `isOpen` (boolean), `message` (textul afișat), `onConfirm` și `onCancel`. Modalul trebuie

să blocheze interacțiunea cu restul paginii prin overlay și să se închidă la apăsarea tastei `Escape`.

Indicație: ascultarea tastei `Escape` se face cu `useEffect` care adaugă un event listener pe `document`, cu cleanup explicit pentru a-l elimina la demontare.

10.3. Exercițiul 3 – Validare client-side completă

Implementați validarea client-side preventivă descrisă în secțiunea „Validarea pe două niveluri". Adăugați și o validare „live": pe măsură ce utilizatorul tastează, erorile din câmpurile deja completate dispar automat. Indicație: într-un `useEffect` care depinde de `[name, price, stock]`, curățați câmpul corespunzător din state-ul `errors` dacă valoarea este acum validă.

Laborator 10: Organizarea aplicației. Routing, custom hooks și structura de proiect

1. Obiectivul laboratorului

Aplicația React construită până acum este, din punct de vedere funcțional, completă: utilizatorul poate vizualiza, adăuga, modifica și șterge produse, cu feedback consistent prin notificări. Din punct de vedere arhitectural, însă, aplicația este monolitică – toată logica rezidă în `App.jsx`, care a devenit un fișier mare cu mai multe variabile de stare, mai multe callback-uri și o structură de tip „switch" care comută între moduri (listă, formular de creare, formular de editare).

Laboratorul curent abordează exact această problemă. Se va transforma aplicația într-un SPA profesional: pagini distincte accesate prin URL-uri proprii (lista produselor la `/products`, detalii la `/products/:id`, formular de creare la `/products/new`, formular de editare la `/products/:id/edit`), navigare cu butoanele de browser (înapoi/înainte) funcționale, posibilitatea de a salva la favorite o pagină specifică sau de a o partaja prin link. Pentru aceasta se va folosi React Router, biblioteca standard pentru rutare în aplicații React. Se va extrage apoi logica de comunicare cu serverul în custom hooks reutilizabile, se va reorganiza proiectul pe directoare cu rol bine definit și se va încheia cu producerea unui build optimizat pentru livrare în producție.

La final, aplicația va fi funcțional echivalentă cu cea din laboratorul anterior, dar structurată într-un mod care suportă creșterea organică spre o aplicație mai mare.

La finalul laboratorului, cititorul ar trebui să fie capabil:

- să explice ce este routing-ul într-un SPA și cum diferă de modelul tradițional client-server;
- să instaleze și să configureze React Router pentru o aplicație cu mai multe pagini;
- să definească rute statice și rute cu parametri dinamici;
- să folosească corect componentele `Link` și `NavLink` pentru navigare declarativă;
- să utilizeze hook-urile `useNavigate`, `useParams` și `useLocation`;
- să extragă logica de comunicare cu serverul în custom hooks reutilizabile;
- să organizeze un proiect React pe directoare cu rol bine definit;
- să producă un build de producție și să înțeleagă cum se livrează aplicația către un server web.

2. Routing într-un SPA

Într-o aplicație web tradițională, cu randare pe server, fiecare URL corespunde unei pagini distincte servită de server. Când utilizatorul apasă un link sau introduce un URL în bara de adrese, browserul face o cerere către server, primește înapoi o pagină HTML completă și o afișează. Server-ul cunoaște toate rutele aplicației și știe să le mapeze pe controllere care produc HTML. [9, 40]

Într-un SPA, browserul nu mai face cereri pentru pagini noi. Există o singură pagină HTML – `index.html` – livrată o singură dată la prima vizită, iar conținutul vizibil se schimbă în funcție de ce generează React. Server-ul nu mai cunoaște rutele aplicației, ci doar URL-ul de bază. Logica de a decide ce se afișează la fiecare URL trece în client.

Totuși, utilizatorii așteaptă comportamente specifice de browser care erau disponibile în modelul tradițional: să poată salva la favorite o pagină de produs (URL-ul trebuie să identifice produsul), să dea înapoi/înainte cu butoanele browserului (URL-urile vizitate trebuie să fie reținute în istoric), să copieze un link și să-l deschidă într-un alt tab (deschiderea acelui URL trebuie să-l ducă direct pe utilizator în starea respectivă). Toate acestea presupun ca URL-ul să reflecte starea aplicației – chiar dacă, tehnic, navigarea nu mai produce cereri către server.

2.1. Cum funcționează routing-ul client-side

Mecanismul fundamental al routing-ului în SPA-uri se bazează pe API-ul History al browserului (`history.pushState`, `history.replaceState`), care permite codului JavaScript să modifice URL-ul afișat în bara de adrese fără a declanșa o cerere către server. Browserul actualizează URL-ul și adaugă o intrare în istoric – exact ca o navigare normală – dar pagina nu se reîncarcă. [9, 11]

React Router este o bibliotecă care folosește History API pentru a oferi un model declarativ, în stil React, de definire a rutelor. Programatorul descrie un arbore de rute – „la URL-ul `/products` randează componenta `ProductsListPage`, la URL-ul `/products/:id` randează `ProductDetailPage`" – iar biblioteca se ocupă de restul: ascultă schimbările URL-ului, decide ce componentă este activă, oferă acces la parametrii din URL.

3. Instalarea și configurarea React Router

Dintr-un terminal poziționat în directorul `products-ui`, instalăm pachetul:

```
npm install react-router-dom
```

Pachetul `react-router-dom` conține integrarea cu DOM-ul browserului. Există și `react-router-native` pentru React Native, dar nu ne preocupă în acest îndrumar.

3.1. BrowserRouter – activarea routing-ului

Punctul de plecare este componenta `<BrowserRouter>`, care trebuie să încadreze întreaga aplicație. Ea activează ascultarea evenimentelor History API și pune la dispoziția componentelor din interior accesul la URL-ul curent. Locul natural este `src/main.jsx`:

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";
import { BrowserRouter } from "react-router-dom";
import "./index.css";
import App from "./App.jsx";

createRoot(document.getElementById("root")).render(
  <StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </StrictMode>
);
```

După această modificare, aplicația are routing activ – dar deocamdată nu folosește nimic. Pasul următor este definirea rutelor în interiorul lui App.

3.2. Routes și Route – definirea rutelor

Se vor defini rutele în `App.jsx` printr-o pereche `<Routes> / <Route>`. Aplicația va avea cinci rute:

- **`/`** – redirecționează la `/products` (pagina principală)
- **`/products`** – lista produselor
- **`/products/new`** – formularul de creare a unui produs nou
- **`/products/:id`** – pagina de detaliu a unui produs (cu parametru dinamic `id`)
- **`/products/:id/edit`** – formularul de editare al produsului respectiv

În plus, se va adăuga o pagină 404 pentru orice URL care nu se potrivește cu cele de mai sus. Configurarea integrală arată astfel:

```
// src/App.jsx

import { Routes, Route, Navigate } from "react-router-dom";
import Layout from "./components/Layout.jsx";
import ProductsListPage   from "./pages/ProductsListPage.jsx";
import ProductDetailPage  from "./pages/ProductDetailPage.jsx";
import ProductCreatePage  from "./pages/ProductCreatePage.jsx";
import ProductEditPage    from "./pages/ProductEditPage.jsx";
import NotFoundPage       from "./pages/NotFoundPage.jsx";

function App() {
  return (
    <Layout>
      <Routes>
        <Route path="/"
```

```
                element={<Navigate to="/products" replace />} />
          <Route path="/products"
                element={<ProductsListPage />} />
          <Route path="/products/new"
                element={<ProductCreatePage />} />
          <Route path="/products/:id"
                element={<ProductDetailPage />} />

          <Route path="/products/:id/edit"
                element={<ProductEditPage />} />
          <Route path="*"
                element={<NotFoundPage />} />
        </Routes>
      </Layout>
    );
  }

  export default App;
```

`<Routes>` este containerul care decide, în funcție de URL-ul curent, care dintre rutele interioare se potrivește; randează doar pe aceea.

`<Route path="..." element={...} />` asociază un șablon de URL cu un element React de generat. Pattern-ul `:id` în `/products/:id` marchează un parametru dinamic – pe acea poziție se acceptă orice valoare, care va fi disponibilă în componenta generată prin hook-ul `useParams`.

`<Navigate to="..." replace />` este o componentă utilitară care, atunci când este randată, redirecționează imediat browserul către URL-ul indicat. Atributul `replace` înlocuiește intrarea în istoric, astfel încât utilizatorul nu poate ajunge înapoi la / apăsând butonul de back – îl trimite direct la cea anterioară lui /.

Ruta cu `path="*"` este o rută „catch-all”: se potrivește cu orice URL care nu a fost preluat de rutele anterioare. Plasarea ei la finalul listei este obligatorie, pentru că `<Routes>` alege prima rută care se potrivește, în ordinea declarării.

4. Componenta Layout

Se poate observa că, în configurarea de mai sus, toate rutele sunt încadrate într-o componentă `Layout`. Acesta este un pattern uzual: layout-ul conține elementele comune ale tuturor paginilor (antetul cu titlul aplicației, meniul de navigare, eventual un footer), iar conținutul specific paginii curente este randat în interior. [9, 45]

Componenta `Layout` primește copii (*children*) prin props:

```
// src/components/Layout.jsx

import { Link, NavLink } from "react-router-dom";
import "./Layout.css";

function Layout({ children }) {
```

```
    return (
      <div className="app">
        <header>
          <Link to="/" className="logo">
            Catalog produse
          </Link>

          <nav>
            <NavLink
              to="/products"
              className={({ isActive }) =>
                isActive ? "nav-link active" : "nav-link"
              }
            >
              Listă
            </NavLink>
            <NavLink
              to="/products/new"
              className={({ isActive }) =>
                isActive ? "nav-link active" : "nav-link"
              }
            >
              + Adaugă
            </NavLink>
          </nav>
        </header>

        <main>{children}</main>

        <footer>
          Programare Web – aplicație de laborator
        </footer>
      </div>
    );
  }

  export default Layout;
```

4.1. Link – navigare declarativă

`<Link>` este componenta pentru navigare în arborele JSX. Vizual produce un element `<a>` (deci este accesibil, are stilurile implicite de link, suportă click cu Ctrl pentru a deschide în tab nou), dar interceptează click-ul obișnuit și folosește History API pentru a schimba URL-ul fără a recarca pagina. [9]

Atributul `to` primește URL-ul țintă. Poate fi absolut (`to="/products"`) sau relativ (`to="new"`, aplicat la URL-ul curent).

> **Atenție**
>
> Tag-urile `<a>` obișnuite în JSX nu trebuie folosite pentru navigarea internă a aplicației. Ele produc o reîncărcare completă a paginii, ceea ce distruge starea SPA-ului (pierdem

state-ul, refacem toate apelurile HTTP, animațiile dispar). `<a>` rămâne valid pentru linkuri externe (către alte site-uri) și pentru ancore (`href="#sectiune"`).

4.2. NavLink – stilizarea linkului activ

`<NavLink>` este o variantă specializată a lui `<Link>` care primește automat informația dacă URL-ul curent corespunde linkului. Acest lucru permite stilizarea condițională a linkului activ – o componentă esențială a oricărui meniu de navigare.

Sintaxa folosită mai sus – `className={({ isActive }) => ...}` – este forma callback a lui `className`. React Router îi trimite un obiect care conține flag-ul `isActive`; se va întoarce un string diferit în funcție de valoare. Pentru rute mai complexe, obiectul mai conține `isPending` (`true` cât timp se încarcă datele dintr-un loader). [9]

Stilizarea linkului activ se face în `Layout.css`:

```
.nav-link {
  padding: 8px 16px;
  color: #4a5568;
  text-decoration: none;
  border-radius: 4px;
}

.nav-link:hover {
  background: #edf2f7;
}

.nav-link.active {
  background: #2c5282;
  color: white;
}
```

5. Hook-urile useParams și useNavigate

5.1. useParams – citirea parametrilor de URL

Ruta `/products/:id` conține parametrul dinamic `id`. În interiorul componentei `ProductDetailPage`, valoarea acestuia se extrage cu hook-ul `useParams`:

```
// src/pages/ProductDetailPage.jsx

import { useEffect, useState } from "react";
import { useParams, Link } from "react-router-dom";
import { productApi } from "../api/productApi";

function ProductDetailPage() {
  const { id } = useParams();
  const [product, setProduct] = useState(null);
  const [loading, setLoading] = useState(true);
  const [error, setError]     = useState(null);
```

```
  useEffect(() => {
    productApi.getById(id)
      .then(setProduct)
      .catch(setError)
      .finally(() => setLoading(false));
  }, [id]);

  if (loading) return <p>Se încarcă...</p>;
  if (error)   return <p>Eroare: {error.message}</p>;


  return (
    <article className="product-detail">
      <h2>{product.name}</h2>
      <p>Preț: {product.price.toFixed(2)} lei</p>
      <p>Stoc: {product.stock} buc</p>
      <Link to={`/products/${id}/edit`}>Editează</Link>
    </article>
  );
}

export default ProductDetailPage;
```

Observație

`useParams()` returnează un obiect cu toate parametrii rutei active. Pentru `/products/:id`, obiectul este `{ id: "5" }` când URL-ul curent este `/products/5`.

Valorile sunt mereu șiruri de caractere, chiar dacă vizual arată ca numere – React Router nu poate ști ce tip așteaptă aplicația. Pentru un backend care primește id-ul în URL, șirul funcționează direct; pentru un `id` folosit în comparație numerică, este nevoie de `Number(id)`.

Array-ul de dependințe `[id]` al `useEffect` este esențial. Dacă utilizatorul navighează de la `/products/5` la `/products/7` fără să iasă din componentă (de exemplu, prin a apăsa un link din pagina de detaliu către alt produs), `id` se schimbă, dar componenta nu se demontează – React Router doar îi schimbă props-ul. Datorită dependinței, `useEffect` detectează schimbarea și încarcă noile date. Fără dependință, s-ar vedea în continuare produsul vechi.

5.2. useNavigate – navigare programatică

`<Link>` acoperă cazurile de navigare declarativă – utilizatorul apasă efectiv pe un link vizibil. Dar există situații în care este nevoie de navigare programatică: după o operație de salvare reușită, după autentificare, după ștergere. Pentru aceasta există hook-ul `useNavigate`. [9]

```
// src/pages/ProductCreatePage.jsx

import { useNavigate } from "react-router-dom";
import { productApi } from "../api/productApi";
import CreateProductForm from "../components/CreateProductForm.jsx";

function ProductCreatePage() {
  const navigate = useNavigate();

  const handleSuccess = (newProduct) => {
    navigate(`/products/${newProduct.id}`);
  };


  return (
    <div>
      <h2>Adaugă produs nou</h2>
      <CreateProductForm
        onSuccess={handleSuccess}
        onCancel={() => navigate("/products")}
      />
    </div>
  );
}

export default ProductCreatePage;
```

Apelul `useNavigate()` returnează o funcție care, la apelare, schimbă URL-ul. Apelul `navigate("/products")` mută browserul la URL-ul absolut indicat (rerandând componenta corespunzătoare). Funcția suportă și un al doilea argument cu opțiuni:

- `navigate("/products", { replace: true })` – înlocuiește intrarea curentă în istoric (util după login sau redirect).
- `navigate(-1)` – mută înapoi cu o intrare în istoric (echivalentul lui `history.back()`).
- `navigate("/products", { state: { fromCreate: true } })` – transmite un obiect de stare către pagina destinație, accesibil acolo prin hook-ul `useLocation`. Util pentru a transporta informație suplimentară între pagini fără a o pune în URL.

6. Pagina 404

Ruta `<Route path="*" element={<NotFoundPage />} />` definită în Routes captează orice URL care nu se potrivește cu rutele specifice. Componenta `NotFoundPage` este un punct natural pentru a oferi utilizatorului o experiență prietenoasă atunci când rătăcește către un URL inexistent. [9]

```
// src/pages/NotFoundPage.jsx

import { Link, useNavigate, useLocation } from "react-router-dom";

function NotFoundPage() {
```

```
    const navigate = useNavigate();
    const location = useLocation();

    return (
      <div className="not-found">
        <h2>404 – Pagina nu există</h2>
        <p>
          URL-ul <code>{location.pathname}</code> nu corespunde
          niciunei pagini din aplicație.
        </p>
        <div className="actions">
          <button onClick={() => navigate(-1)}>Înapoi</button>


          <Link to="/products" className="btn-primary">
            La catalogul de produse
          </Link>
        </div>
      </div>
    );
  }

  export default NotFoundPage;
```

> **Observație**
>
> `useLocation` este un hook care returnează obiectul `Location` curent, cu câmpurile `pathname`, `search` (querystring-ul), `hash` (ancora), `state` (obiectul transmis la navigare). Aici se folosește pentru a afișa utilizatorului URL-ul exact pe care l-a căutat – un detaliu mic, dar care îi dă o senzație de control asupra a ceea ce se întâmplă.

7. Custom hooks – extracția logicii de date

În laboratoarele anterioare se observă un pattern care se repetă: pentru a încărca o resursă de pe server, se definesc trei variabile de stare (data, loading, error), se apelează `productApi` într-un `useEffect`, se tratează cele trei căi (succes, eroare, finalizare). Acest pattern apare acum în mai multe locuri:

- `ProductsListPage` încarcă lista de produse;
- `ProductDetailPage` încarcă un singur produs;
- `ProductEditPage` încarcă produsul ce urmează a fi editat.

Repetiția este simptomul unei abstracții ratate. React oferă mecanismul pentru această abstracție: custom hooks – funcții JavaScript obișnuite care, prin convenție, încep cu prefixul `use` și pot apela în interiorul lor alte hook-uri (`useState`, `useEffect` etc.). Extragerea logicii de încărcare într-un hook propriu reduce codul componentelor. [7, 32]

7.1. Hook-ul useProducts

```
// src/hooks/useProducts.js

import { useEffect, useState } from "react";
import { productApi } from "../api/productApi";

export function useProducts() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading]   = useState(true);
  const [error, setError]       = useState(null);

  const reload = () => {
    setLoading(true);
    setError(null);
    productApi.getAll()
      .then(setProducts)
      .catch(setError)
      .finally(() => setLoading(false));
  };

  useEffect(() => {
    reload();
  }, []);

  return { products, loading, error, reload };
}
```

Observație

Numele începe cu `use` – aceasta nu este pură convenție, este o regulă a React: linter-ul folosește prefixul pentru a recunoaște hook-urile și a verifica regulile lor. Fără prefix, `useState` și `useEffect` ar produce eroare de execuție.

Funcția exportată este o funcție obișnuită, nu o componentă. Întoarce un obiect cu câmpurile relevante pentru consumator. Decizia de a întoarce un obiect (în loc de un array) face apelul mai descriptiv și permite consumatorului să destructureze doar câmpurile pe care le folosește.

Funcția `reload` este expusă în obiectul returnat – permite consumatorului să declanșeze o nouă încărcare (de exemplu, după ce s-a creat un produs nou).

7.2. Componenta ProductsListPage devine mai simplă

```
// src/pages/ProductsListPage.jsx

import { Link } from "react-router-dom";
import { useProducts } from "../hooks/useProducts";
import ProductCard from "../components/ProductCard.jsx";
import Spinner from "../components/Spinner.jsx";
```

```
import ErrorMessage from "../components/ErrorMessage.jsx";

function ProductsListPage() {
  const { products, loading, error, reload } = useProducts();

  if (loading) return <Spinner />;
  if (error)   return <ErrorMessage error={error} onRetry={reload} />;
  if (products.length === 0) {
    return (
      <div className="empty-state">
        <p>Nu există produse înregistrate.</p>
        <Link to="/products/new">Adaugă primul produs</Link>
      </div>
    );
  }

  return (
    <div className="product-list">
      {products.map(p => (
        <ProductCard key={p.id} product={p} onDelete={...} />
      ))}
    </div>
  );
}

export default ProductsListPage;
```

Toată logica de comunicare cu serverul a dispărut din componentă. A rămas în `useProducts`, care este reutilizabil oriunde altundeva ar fi nevoie de lista de produse. Componenta a devenit pur prezentațională – primește date, le afișează.

7.3. Hook-ul useProduct

Pentru un singur produs, hook-ul este analog, dar primește id-ul ca argument și are o particularitate – cazul 404 (produs inexistent) trebuie tratat distinct [7]:

```
// src/hooks/useProduct.js

import { useEffect, useState } from "react";
import { productApi } from "../api/productApi";

export function useProduct(id) {
  const [product, setProduct]   = useState(null);
  const [loading, setLoading]   = useState(true);
  const [notFound, setNotFound] = useState(false);
  const [error, setError]       = useState(null);

  useEffect(() => {
    setLoading(true);
    setNotFound(false);
    setError(null);

    productApi.getById(id)
      .then(setProduct)
```

```
      .catch(err => {
        if (err.response?.status === 404) {
          setNotFound(true);
        } else {
          setError(err);
        }
      })
      .finally(() => setLoading(false));
  }, [id]);

  return { product, loading, notFound, error };
}
```

Câmpul `notFound` este expus separat de `error`. Distincția este importantă: un produs inexistent este o situație previzibilă (utilizatorul poate introduce manual un id greșit, sau poate ajunge prin link de la un email vechi), care merită un mesaj prietenos. Un eșec real al cererii (server inaccesibil, rețea) este o situație neașteptată care merită un mesaj diferit.

Componenta `ProductDetailPage` refactorizată:

```
function ProductDetailPage() {
  const { id } = useParams();
  const { product, loading, notFound, error } = useProduct(id);

  if (loading)  return <Spinner />;
  if (notFound) return <NotFoundProduct id={id} />;
  if (error)    return <ErrorMessage error={error} />;

  return (
    <article className="product-detail">
      <h2>{product.name}</h2>
      <p>Preț: {product.price.toFixed(2)} lei</p>
      <p>Stoc: {product.stock} buc</p>
      <Link to={`/products/${id}/edit`}>Editează</Link>
    </article>
  );
}
```

8. Structura proiectului

Pe măsură ce au fost adăugate funcționalități, numărul de fișiere a crescut. Pentru a păstra proiectul organizat, se pot grupa pe directoare cu rol bine definit:

```
src/
├── api/                        Comunicarea cu serverul
│   └── productApi.js
├── components/                 Componente vizuale reutilizabile
│   ├── ConfirmModal.jsx
│   ├── CreateProductForm.jsx
│   ├── EditProductForm.jsx
│   ├── ErrorMessage.jsx
│   ├── Layout.jsx
│   ├── Layout.css
│   ├── ProductCard.jsx
```

```
│   ├── Spinner.jsx
│   ├── StockBadge.jsx
│   └── Toast.jsx
├── hooks/                      Custom hooks
│   ├── useProduct.js
│   └── useProducts.js
├── pages/                      Componente legate de o rută
│   ├── NotFoundPage.jsx
│   ├── ProductCreatePage.jsx
│   ├── ProductDetailPage.jsx
│   ├── ProductEditPage.jsx
│   └── ProductsListPage.jsx
├── App.css
├── App.jsx                     Definirea rutelor
├── index.css
└── main.jsx                    Punctul de intrare
```

8.1. Justificarea separării

Fiecare director are un rol distinct, ușor de articulat:

- **`api/`** conține modulele care se ocupă de comunicarea HTTP cu serverul. Pe măsură ce aplicația crește, fiecare resursă din API (produse, comenzi, utilizatori) va avea propriul fișier (`orderApi.js`, `userApi.js`).
- **`components/`** conține componente vizuale reutilizabile, fără rută proprie. Pot fi atomice (`Spinner`, `StockBadge`), pot fi formulare (`CreateProductForm`), pot fi widget-uri (`Toast`).
- **`hooks/`** conține custom hooks. Numele fișierului începe cu use (`useProducts.js`, `useProduct.js`), pentru consistență.
- **`pages/`** conține componentele care corespund unei rute în React Router. Sunt, în mare, „orchestratori”: aduc date prin hooks, le pasează componentelor din `components/`, ascultă evenimente. Sunt puține și greu reutilizabile (au rost doar la URL-ul lor).

> **Alternative**
>
> Aceasta nu este singura convenție posibilă. O alternativă populară este *feature folders*: fiecare director conține totul ce ține de o resursă (de exemplu, `src/products/` conține api, componente, hooks, pagini, totul în interior). Pentru aplicații foarte mari, această structură evită ca directorul `components/` să devină inutilizabil prin volumul de fișiere. Pentru aplicații de talie mică spre medie, separarea pe roluri (cea de mai sus) rămâne preferată pentru claritate didactică.

8.2. App.jsx devine un fișier scurt

Cu această structură, `App.jsx` conține doar definirea rutelor, fără logică:

```
// src/App.jsx (final)

import { Routes, Route, Navigate } from "react-router-dom";
import Layout from "./components/Layout.jsx";
```

```
import ProductsListPage  from "./pages/ProductsListPage.jsx";
import ProductDetailPage from "./pages/ProductDetailPage.jsx";
import ProductCreatePage from "./pages/ProductCreatePage.jsx";
import ProductEditPage   from "./pages/ProductEditPage.jsx";
import NotFoundPage      from "./pages/NotFoundPage.jsx";

function App() {
  return (
    <Layout>
      <Routes>
        <Route path="/"
               element={<Navigate to="/products" replace />} />
        <Route path="/products"
               element={<ProductsListPage />} />
        <Route path="/products/new"
               element={<ProductCreatePage />} />
        <Route path="/products/:id"
               element={<ProductDetailPage />} />
        <Route path="/products/:id/edit"
               element={<ProductEditPage />} />
        <Route path="*"
               element={<NotFoundPage />} />
      </Routes>
    </Layout>
  );
}

export default App;
```

9. Build de producție

Până acum aplicația a fost executată prin `npm run dev` – serverul de dezvoltare Vite, cu hot module replacement, mesaje de eroare detaliate, fără minimizare. Acest mod este optim pentru dezvoltare, dar absolut nepotrivit pentru livrarea către utilizatori finali: bundle-ul este mare, codul este lizibil, source maps-urile sunt expuse. [8, 44]

Pentru livrare avem nevoie de un *build de producție*:

```
npm run build
```

Comanda execută `vite build`, care:

- transformă JSX-ul și sintaxa modernă JavaScript la JavaScript compatibil cu browserele țintă;
- aplică tree-shaking (elimină codul nefolosit din biblioteci);
- minimizează JS-ul și CSS-ul (elimină spațiile, scurtează nume de variabile);
- împarte codul în chunks (lazy loading pentru rute, dacă este configurat);
- calculează hash-uri pentru fiecare fișier și le include în nume (cache busting);
- produce, în final, un director `dist/` cu HTML, JS, CSS și asset-uri.

Conținutul tipic al directorului `dist/` după build:

```
dist/
├── assets/
│   ├── index-B7id5rbw.js      ← bundle-ul aplicației (minificat)
│   ├── index-Df7vJ29x.css     ← stilurile (minificate)
│   └── react-DqzIvhsl.js      ← React, separat (cache-uibil)
├── index.html
└── vite.svg
```

Hash-urile din nume (`B7id5rbw`, `Df7vJ29x`) sunt calculate din conținutul fișierului. La fiecare modificare, hash-ul se schimbă, deci numele fișierului se schimbă, iar browserul descarcă noua versiune. La fișierele care nu s-au modificat, hash-ul rămâne identic, browserul le servește din cache. Acest mecanism, numit *cache busting*, permite cache headers foarte agresive (asset-uri valabile un an) fără risc de a servi versiuni învechite.

9.1. Previzualizarea build-ului local

Vite oferă o comandă pentru a verifica local că build-ul funcționează corect:

```
npm run preview
```

Comanda pornește un mic server static (implicit pe portul 4173) care servește directorul `dist/`. Este o simulare a serverului de producție, suficient pentru a verifica vizual că aplicația merge ca atunci când a fost livrată. Nu se folosește în producție reală; pentru aceasta, `dist/` se servește cu un server web profesional. [8]

9.2. SPA fallback

Un detaliu subtil, dar care produce eșecuri spectaculoase dacă nu este înțeles: pentru ca rutele SPA să funcționeze și după *refresh manual* sau *deschidere directă a URL-ului* (de exemplu, `https://aplicatie.example.com/products/5/edit`), serverul web care servește `dist/` trebuie configurat să returneze `index.html` pentru orice rută necunoscută.

Problema concretă: utilizatorul navighează în SPA până la `/products/5/edit`. URL-ul se schimbă, dar nu există o cerere către server. Dacă apasă F5 (refresh), browserul trimite o cerere reală către `https://aplicatie.example.com/products/5/edit`. Serverul nu cunoaște această rută – ea există doar în React Router. Implicit, returnează 404.

Soluția este SPA fallback: serverul răspunde cu `index.html` pentru orice rută care nu corespunde unui fișier real în `dist/`. Apoi, în browser, React Router preia URL-ul curent și randează componenta potrivită.

9.2.1. Exemplu de configurare nginx

Pentru un server nginx care servește aplicația, configurarea minimă este:

```
server {
    listen 80;
    server_name aplicatie.example.com;
    root /var/www/products-ui/dist;
    index index.html;
```

```
    # Asset-urile cu hash în nume: cache agresiv
    location /assets/ {
        expires 1y;
        add_header Cache-Control "public, immutable";
        try_files $uri =404;
    }

    # Orice altă rută: dacă fișierul nu există, livrează index.html
    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Directiva esențială este `try_files $uri $uri/ /index.html` în `location /`. Nginx încearcă, în ordine: să găsească un fișier cu calea cerută, un director cu acea cale, iar dacă niciunul nu există, returnează `index.html`. Acesta este SPA fallback-ul.

> **Notă**
>
> Comanda `npm run preview` a lui Vite face deja acest fallback automat, motiv pentru care, în dezvoltare, problema nu apare. Aceasta apare doar la livrare cu un server web generic (nginx, Apache, IIS) care, implicit, nu cunoaște conceptul de SPA.

10. Direcții pentru aprofundare

Modulul React acoperă esențialul pentru construirea de SPA-uri profesionale care consumă un API REST. Pentru aplicații reale de talie mare, direcții de aprofundare sunt:

Autentificarea – integrarea cu mecanismul JWT: pagină de login, transmiterea automată a token-ului în antetul `Authorization` (prin interceptori `axios`), guard-uri pe rute pentru a proteja zonele care necesită utilizator autentificat.

State management global – pentru aplicații cu state distribuit pe multe componente non-înrudite, Context API (intern React) sau biblioteci precum Redux Toolkit, Zustand, Jotai oferă soluții uniforme.

Server state cu TanStack Query (React Query) – pentru gestionarea inteligentă a cache-ului datelor de pe server (refetch automat, optimistic updates, invalidare). Este o evoluție naturală a custom hook-urilor din acest laborator, dar cu un set de funcționalități semnificativ mai bogat.

Testarea componentelor – cu Vitest și React Testing Library. Testele de componentă verifică randarea corectă în diverse stări și răspunsul la evenimente, completând testele de backend deja existente.

Framework-uri full-stack – Next.js, Remix (acum unificat cu React Router 7) oferă, pe lângă routing-ul client-side, server-side rendering, server components, optimizări automate. Sunt potrivite pentru aplicații cu cerințe de SEO sau performanță inițială foarte strictă.

11. Exerciții propuse

11.1. Exercițiul 1 – Confirmare de părăsire a formularului

Adăugați protecție împotriva pierderii datelor: dacă utilizatorul a modificat un câmp în `ProductCreatePage` sau `ProductEditPage` și încearcă să navigheze (prin Link sau buton de back), apare un dialog „Aveți modificări nesalvate. Doriți să părăsiți pagina?”. Indicație: `useBlocker` din React Router este mecanismul oficial; alternativ, evenimentul `beforeunload` al browserului pentru a captura și închiderea tab-ului.

11.2. Exercițiul 2 – Lazy loading al rutelor

Pentru a reduce mărimea bundle-ului inițial, transformați paginile în *lazy components*:

```
const ProductsListPage = lazy(() =>
import("./pages/ProductsListPage.jsx"));
```

Și încadrați `<Routes>` într-un `<Suspense fallback={<Spinner />}>`. Verificați în DevTools → Network că, la prima vizită, se descarcă doar codul rutei curente; la navigare către altă rută, se descarcă codul ei separat.

11.3. Exercițiul 3 – Deployment cu nginx în Docker

Construiți un Dockerfile cu două etape: prima etapă rulează `npm install` și `npm run build` pe o imagine Node.js; a doua etapă copiază `dist/` într-o imagine `nginx:alpine`, cu configurația de SPA fallback descrisă în laborator. Rezultatul este un container, gata de pus în producție.

12. Bibliografie recomandată

12.1. Documentație oficială

[1] Spring Team, *Spring Boot Reference Documentation*, VMware Inc. Disponibil la: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

[2] Spring Team, *Spring Framework Documentation*, VMware Inc. Capitolul Web MVC, Validation, Data Access. Disponibil la: https://docs.spring.io/spring-framework/reference/

[3] Spring Team, *Spring Security Reference*, VMware Inc. Disponibil la: https://docs.spring.io/spring-security/reference/

[4] springdoc-openapi contributors, *springdoc-openapi – OpenAPI 3 și Swagger UI pentru Spring Boot*. Documentație oficială a proiectului. Disponibil la: https://springdoc.org/

[5] Spring Team, *Spring Data JPA Reference Documentation*, VMware Inc. Disponibil la: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

[6] Hibernate Team, *Hibernate ORM User Guide*, Red Hat Inc. Disponibil la: https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html

[7] React Team, *React – The library for web and native user interfaces*, Meta Platforms Inc. Disponibil la: https://react.dev (accesat: mai 2026; versiunea curentă: React 19.2)

[8] Y. Evan și colaboratorii, *Vite – Next Generation Frontend Tooling*. Disponibil la: https://vite.dev (accesat: mai 2026; versiunea curentă: Vite 8.x cu bundler Rolldown)

[9] React Router Team, *React Router Documentation*. Disponibil la: https://reactrouter.com (versiunea curentă: v7)

[10] axios contributors, *axios – Promise-based HTTP client*. Disponibil la: https://axios-http.com

[11] Mozilla Foundation, *MDN Web Docs*, referință completă pentru HTML, CSS, JavaScript, HTTP, DOM. Disponibil la: https://developer.mozilla.org

[12] OpenJS Foundation, *Node.js Documentation*. Disponibil la: https://nodejs.org/en/docs

[13] JUnit Team, *JUnit 5 User Guide*. Disponibil la: https://junit.org/junit5/docs/current/user-guide/

[14] Mockito Team, *Mockito Framework Reference*. Disponibil la: https://site.mockito.org

12.2. Standarde și specificații

[15] ECMA International, *ECMAScript® 2024 Language Specification*, Standard ECMA-262, ediția a 15-a, iunie 2024. Disponibil la: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/

[16] WHATWG, *HTML Living Standard*. Disponibil la: https://html.spec.whatwg.org/

[17] WHATWG, *Fetch Standard* (include și specificația CORS). Disponibil la: https://fetch.spec.whatwg.org/

[18] R. Fielding și J. Reschke (eds.), *RFC 9110 – HTTP Semantics*, IETF, iunie 2022. Standardul actual al HTTP. Disponibil la: https://www.rfc-editor.org/rfc/rfc9110

[19] M. Jones, J. Bradley, N. Sakimura, *RFC 7519 – JSON Web Token (JWT)*, IETF, mai 2015. Specificația JWT folosită în Lab 8. Disponibil la: https://www.rfc-editor.org/rfc/rfc7519

[20] Jakarta EE Working Group, *Jakarta Bean Validation Specification*, versiunea 3.0. Specificația folosită de Spring Boot pentru validarea DTO-urilor. Disponibil la: https://jakarta.ee/specifications/bean-validation/3.0/

12.3. Cărți – Java și ecosistemul Spring

[21] C. Walls, *Spring in Action*, ediția a 6-a, Manning Publications, Shelter Island, NY, 2022, ISBN 978-1-6172-9994-5.

[22] J. Long, *Reactive Spring*, self-published, 2020. Disponibil la: https://reactivespring.io

[23] M. Heckler, *Spring Boot: Up and Running – Building Cloud Native Java and Kotlin Applications*, O'Reilly Media, Sebastopol, CA, 2021, ISBN 978-1-4920-7697-1.

[24] L. Spilcă, *Spring Security in Action*, ediția a 2-a, Manning Publications, Shelter Island, NY, 2024, ISBN 978-1-6334-3797-5.

[25] V. Massol, J. Hunter, *Pragmatic Unit Testing in Java with JUnit*, ediția a 3-a, Pragmatic Bookshelf, 2015, ISBN 978-1-9416-2222-9.

[26] J. Bloch, *Effective Java*, ediția a 3-a, Addison-Wesley Professional, 2017, ISBN 978-0-1346-8599-1. Referință generală pentru bunele practici Java.

12.4. Cărți – JavaScript și ecosistemul React

[27] M. Haverbeke, *Eloquent JavaScript: A Modern Introduction to Programming*, ediția a 4-a, No Starch Press, San Francisco, 2024, ISBN 978-1-7185-0410-3. Disponibil gratuit la: https://eloquentjavascript.net

[28] D. Flanagan, *JavaScript: The Definitive Guide*, ediția a 7-a, O'Reilly Media, Sebastopol, CA, 2020, ISBN 978-1-4919-5202-3.

[29] K. Simpson, *You Don't Know JS Yet*, ediția a 2-a, Leanpub, serie completă, 2020–prezent. Disponibil la: https://github.com/getify/You-Dont-Know-JS

[30] Banks, E. Porcello, *Learning React: Modern Patterns for Developing React Apps*, ediția a 2-a, O'Reilly Media, Sebastopol, CA, 2020, ISBN 978-1-4920-5172-5.

[31] R. Wieruch, *The Road to React*, ediția 2024 (actualizată continuu), self-published. Disponibil la: https://www.roadtoreact.com

[32] M. Schwarzmüller, *React Key Concepts: An In-Depth Guide to React's Core Features*, ediția a 2-a, Packt Publishing, Birmingham, 2024, ISBN 978-1-83620-114-7.

[33] T. Kumar, *Fluent React: Build Fast, Performant, and Intuitive Web Applications*, O'Reilly Media, Sebastopol, CA, 2024, ISBN 978-1-0981-3871-7.

12.5. Cărți – arhitectură și design

[34] L. Richardson, M. Amundsen, S. Ruby, *RESTful Web APIs – Services for a Changing World*, O'Reilly Media, Sebastopol, CA, 2013, ISBN 978-1-4493-5806-8.

[35] M. Masse, *REST API Design Rulebook*, O'Reilly Media, 2011, ISBN 978-1-4493-1050-9.

[36] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, teză de doctorat, University of California, Irvine, 2000. Sursa originală a stilului arhitectural REST. Disponibil la: https://ics.uci.edu/~fielding/pubs/dissertation/top.htm

[37] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley Professional, 2003, ISBN 978-0-3211-2521-7.

[38] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley Professional, 2002, ISBN 978-0-3211-2742-6.

[39] R. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Prentice Hall, 2017, ISBN 978-0-1349-4432-5.

12.6. Articole și surse online

[40] M. Fowler, *Single Page Application*, articol din seria de patternuri arhitecturale. Disponibil la: https://martinfowler.com/bliki/SinglePageApplication.html

[41] M. Fowler, *Richardson Maturity Model – Steps toward the glory of REST*. Disponibil la: https://martinfowler.com/articles/richardsonMaturityModel.html

[42] React Team, *React 19 – Official Release Notes*, blog oficial React, decembrie 2024. Disponibil la: https://react.dev/blog/2024/12/05/react-19

[43] React Team, *Sunsetting Create React App*, anunțul oficial de retragere a CRA, februarie 2025. Disponibil la: https://react.dev/blog/2025/02/14/sunsetting-create-react-app

[44] Vite Team, *Announcing Vite 8 – Unified bundling with Rolldown*, martie 2026. Disponibil la: https://vite.dev/blog/announcing-vite-8

[45] Osmani, *Learning Patterns – Patterns for building powerful web apps with vanilla JavaScript and React*, 2021–2024. Disponibil gratuit la: https://www.patterns.dev

[46] *web.dev*, ghid Google pentru dezvoltare web modernă, performanță și accesibilitate. Disponibil la: https://web.dev