

# APLICATII WEB CU SUPORT JAVA

Prof.Dr.Ing. **Liliana Dobrică**

*Universitatea POLITEHNICA Bucuresti  
Facultatea Automatica si Calculatoare*

1

## Agenda cursului

- Java
  - Java modern
  - Aplicatii Spring (continuare)

2

## Java modern

- VAR
- records

## VAR

- Se foloseste *doar pentru declararea variabilelor locale sau cand se initializeaza imediat variabila cu o valoare* pentru că Java poate sa deduca logic tipul acestora
- 
- Sintaxe de felul:  

```
ArrayList<String> lista = new ArrayList<String>();
```
- Sau  

```
Optional<Masina> masina= persoana.getMasina();
```
- Pot fi inlocuite cu
- Sau  

```
var lista = new ArrayList<String>();
```

```
var masina=persoana.getMasina();
```
- Daca Java nu poate deduce logic, programul nu se compileaza

## RECORD

- Un **record** este un tip special de clasa care simplifica implementarea unei clase care are doar atribute care sunt *immutable*. Atributele obiectelor nu se modifica.
- Poate avea metode suplimentare
- Poate avea metode sau atribute statice
- Limitari:
  - Record nu poate mosteni alte clase
  - Alte clase nu pot mosteni record
  - Un atribut al unui record nu poate fi modificat dupa initializarea acestuia

## RECORD

```
import java.util.Objects;
public final class Persoana {
    private final String nume;


    public Person(String nume) {
        this.nume = nume;
    }

    public String nume() {
        return nume;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || !getClass().isAssignableFrom(Persoana.class)) return false;
        Persoana persoana = (Persoana) o;
        return Objects.equals(nume, persoana.nume);
    }

    @Override
    public int hashCode() {
        return Objects.hash(nume);
    }

    @Override
    public String toString() {
        return "Persoana[" + " nume = " + nume + "]";
    }
}
```



```
public record Persoana(
    String nume)
{
    //metode
}
```

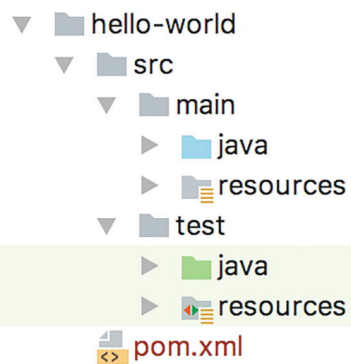
## Aplicatii Spring

- Alte detalii ale procesului de implementare

## Organizarea unui proiect Maven

### Maven

- este fondat de Apache Software Foundation
- este instrumentul principal pentru operatiile de build si gestiunea dependentelor in aplicatiile Java.
- utilizat la impachetarea aplicatiei in fisier jar sau war pentru deploy pe un anumit server
- Defineste structura tipica a unui proiect
- Dependentele sunt declarate in fisierul `pom.xml`



## Separarea in pachete a responsabilitatilor

Separarea in pachete diferite

a responsabilitatilor din proiect

pentru o mai buna intelegere a codului proiectului

## Stabilirea clasei POJO

### Clasa POJO

- POJO = Plain Old Java Object
- Clasa Java care nu este restrictionata de nici un framework
- Modeleaza datele pe care le foloseste aplicatia.
- Este o clasa fara dependente care este definita doar prin attribute si metode.
- Are responsabilitatea de model.
- Se va adauga proiectului un package cu numele `model`.

## Utilizarea de adnotari

Adnotari explicite pentru obiecte -> instruiesc Spring sa creeze si sa adauge la contextul propriu obiectele de tipul claselor adnotate

@Component – are o responsabilitate generala

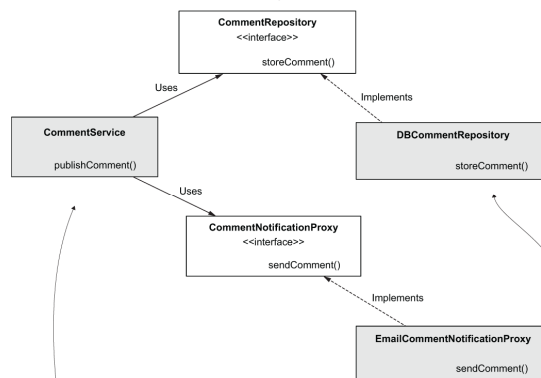
@Service – obiecte care au responsabilitatea de a implementa functionalitati ale aplicatiei

@Repository – obiecte care gestioneaza persistenta datelor

Adnotarile nu se pun pe interfete, ci pe clasele pentru care Spring trebuie sa creeze obiecte pe care sa le adauge la contextul propriu.

## Utilizarea de adnotari

Interfetele sunt abstracte.  
Nu se pun adnotari pe interfete

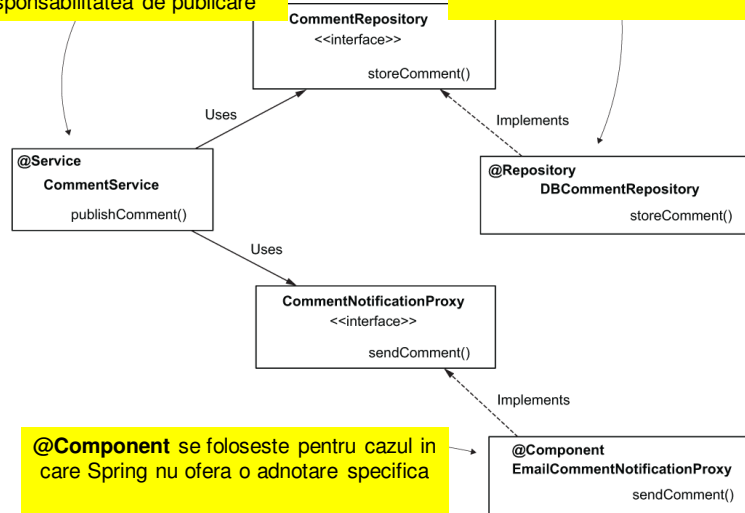


Adnotarile se pun pe aceste clase

## Utilizarea de adnotari

**@Service** definește o componentă Spring cu responsabilitatea de publicare

**@Repository** definește o componentă Spring cu responsabilitate de stocare



**@Component** se folosește pentru cazul în care Spring nu oferă o adnotare specifică

Aplicatii Web cu Suport Java, sem.I, 2024-2025

13

13

## Aplicatie web cu Spring MVC

Adaugarea unei pagini web aplicatiei generate

1. Se scrie documentul HTML care sa fie afisat in browser
2. Se scrie un controller cu actiunea de afisare a documentului HTML



### 1. Scrie documentul HTML

```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Home Page</title>
</head>
<body>
  <h1>Welcome!</h1>
</body>
</html>
    
```



### 2. Scrie un controller cu actiunea de afisare a documentului HTML

```

@Controller
public class MainController {
    @RequestMapping("/home")
    public String home() {
        return "home.html";
    }
}
    
```

Fisierul HTML se adauga in folderul `resources/static` a proiectului

Aplicatii Web cu Suport Java, sem.I, 2024-2025

14

14

## Aplicatie web cu Spring MVC

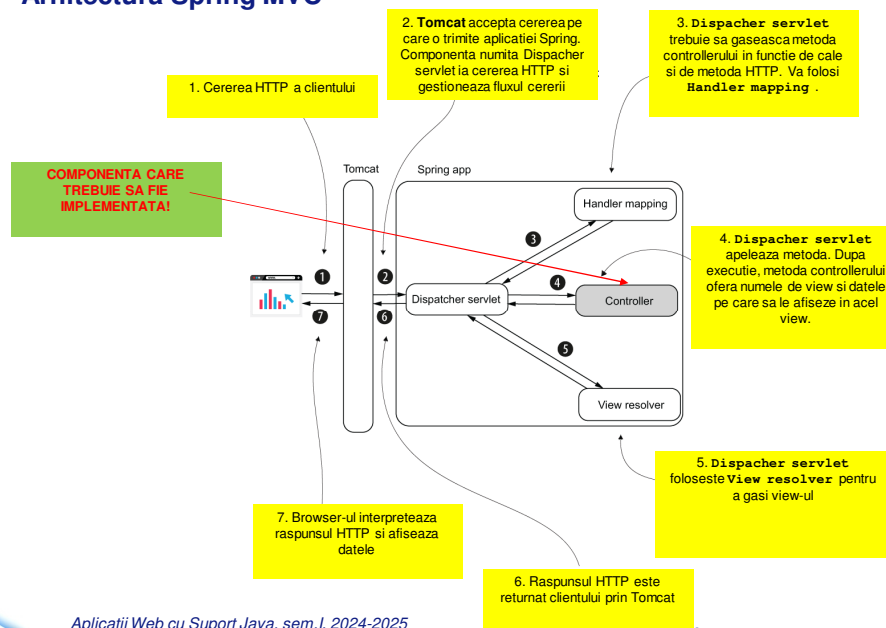
### Definitia clasei controller

1. Clasa este adnotata cu `@Controller`
2. Adnotarea `@RequestMapping ("/home")` se utilizeaza pentru a asocia actiunea din metoda cu calea specificata in browser
3. Metoda returneaza numele fisierului care contine detaliile de afisat in browser

Cu aplicatia in executie, se introduce in browser URL `http://localhost:8080/home`

Browserul interpreteaza si afiseaza continutul documentului HTML primit de la backend ca raspuns la cerere

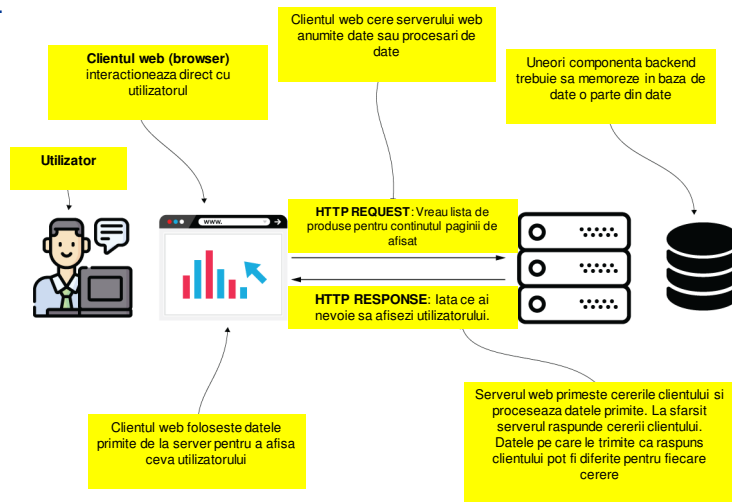
## Arhitectura Spring MVC





## Utilizarea unui template engine Tymeleaf pentru un view dinamic

Pentru o cerere a utilizatorului in browser care devine cerere HTTP, aplicatia primeste anumite date, le prelucreaza si trimite inapoi raspunsul HTTP care se va afisa in browser.

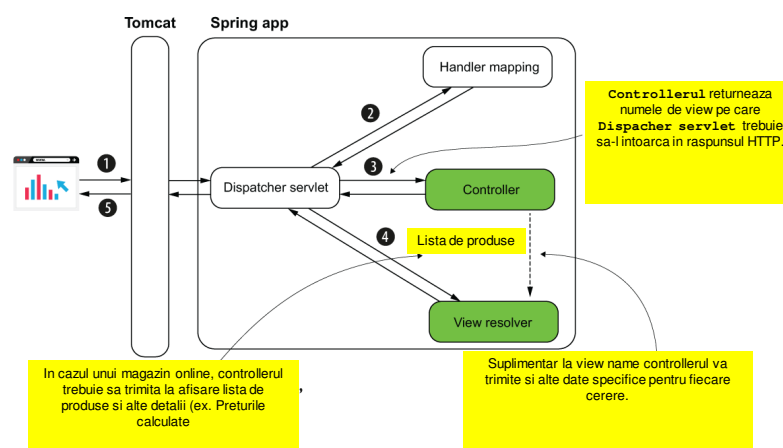


Aplicatii Web cu Suport Java, sem.I, 2024-2025

17

17

## Fluxul Spring in mod dinamic



Aplicatii Web cu Suport Java, sem.I, 2024-2025

18

18

## Utilizarea unui template engine Thymeleaf pentru un view dinamic

Fisierul `pom.xml` contine dependenta de **Thymeleaf**

Dependenta ofera un template engine care permite adaugarea cu usurinta a datelor de la controller la un view care afiseaza intr-un anumit mod aceste date.

```
@Controller  
public class MainController {  
  
    @RequestMapping("/home")  
    public String home(Model pagina) {  
        pagina.addAttribute("username", "Lucia");  
        pagina.addAttribute("color", "red");  
        return "home.html";  
    }  
}
```

1 Actiunea controllerului are asignata calea cererii HTTP

2 Metoda controllerului are un parametru de tip Model care pastreaza datele pe care controllerul le trimite catre view

4 Adaugarea datelor obiectului de tip Model  
`addAttribute(cheie_atribut, valoare_atribut)`

5 Controllerul returneaza view-ul in raspunsul HTTP

Aplicatii Web cu Suport Java, sem.I, 2024-2025

19

19

## Utilizarea unui template engine Thymeleaf pentru un view dinamic

Definirea unui view in fisierul `home.html`  
se adauga fisierul `home.html` in folderul `resources/templates`

```
<!DOCTYPE html>
```

```
<html lang="en" xmlns:th="http://www.thymeleaf.org">
```

```
<head>
```

```
<meta charset="UTF-8">
```

```
<title>Home Page</title>
```

```
</head>
```

```
<body>
```

```
<h1>Welcome
```

```
<span th:style="'color:' + ${color}"
```

```
th:text="${username}"></span>!</h1>
```

```
</body>
```

```
</html>
```

`${cheie_atribut}` se ia valoarea  
atributului de la controller pentru view

1 Defineste prefixul "th" pentru Thymeleaf

2 Foloseste prefixul "th" pentru valorile trimise de controller

Aplicatii Web cu Suport Java, sem.I, 2024-2025

20

20

## Utilizarea unui template engine Tymeleaf pentru un view dinamic

Trimiterea informatiilor de la client la server prin cererea HTTP se realizeaza in mai multe moduri:

- Prin parametrii cererii HTTP – parametri de interogare se adauga la expresia URL
  - Volum mic de informatii
- Prin partea header a cererii HTTP
  - Informatiile nu apar in URL
  - Volum mic de informatii
- Prin variabila de cale
  - Daca valoarea care trebuie trimisa este obligatorie pentru cerere
- Prin partea body a cererii HTTP
  - Pentru trimiterea unui volum mare de date
  - In format String sau fisier

## Prin parametrii cererii HTTP

Se aplica pentru:

- Volum mic de date
    - Limita este 2000 caractere
  - Datele trimise sunt optionale
    - Clientul poate sa nu trimita aceste date
  - Cazuri de utilizare
    - In parametrii cererii se definesc criteriile de cautare sau de filtrare
- 
- La implementarea pe server se va tine cont că aceste informatii pot lipsi
  - In clasa Controller, metoda va avea parametrii cu adnotarea @RequestParam
  - Parametrul din cererea HTTP va avea acelasi nume cu numele parametrului metodei

## Prin parametrii cererii HTTP

Exemplu:

```
@Controller
public class MainController {

    @RequestMapping("/home")
    public String home(
        @RequestParam String color,
        Model pagina) {
        pagina.addAttribute("username", "Katy");
        pagina.addAttribute("color", color);
        return "home.html";
    }
}
```

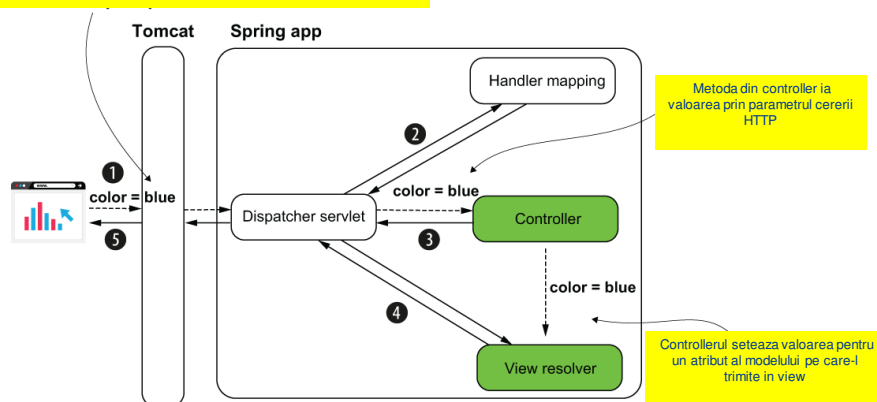
1 Parametrul metodei are același nume cu parametrul din cerere și are adnotarea `@RequestParam`

2 Prin parametrul de tip `Model` controllerul trimite date către view

3 Valoarea parametrului `color` trece de la controller la client  
`addAttribute(cheie_atribut, valoare_atribut)`

## Prin parametrii cererii HTTP

Cientul trimite valoarea parametrului `color` prin parametrul cererii HTTP



Syntaxa din browser va fi completată cu simbolul `?` Urmă de parametru specificat prin perechea cheie-valoare:  
`http://localhost:8080/home?color=blue`  
Pentru o listă de parametri – separarea între parametrii utilizează simbolul `&`:  
`http://localhost:8080/home?color=blue & nume=Ana`

## Prin parametrii cererii HTTP

Exemplu:

```
@Controller
public class MainController {

    @RequestMapping("/home")
    public String home(
        @RequestParam(required = false) String nume,      ❶
        @RequestParam(required = false) String color,
        Model page) {
        page.addAttribute("username", nume);              ❷
        page.addAttribute("color", color);
        return "home.html";
    }
}
```

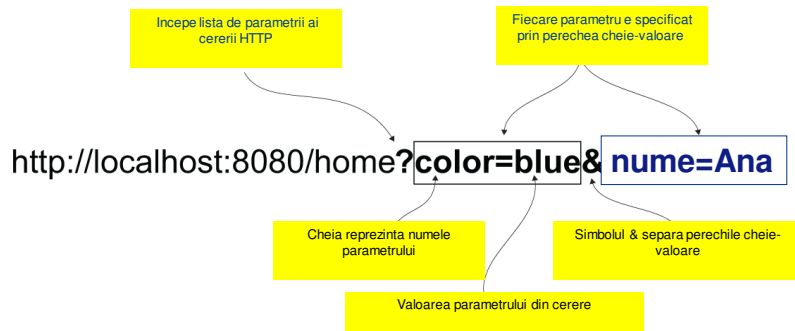
❶ Metoda are un nou parametru "nume" care are adnotarea @RequestParam

❷ Controllerul trimite valoarea parametrului "nume" catre view

In sintaxa cheie=valoare (exemplu color=blue) "cheie" reprezinta numele parametrului, iar valoarea este scrisa dupa simbolul =

25

## Prin parametrii cererii HTTP



26

## Prin variabile de cale

Exemplu:

Cu parametri de cerere  
`http://localhost:8080/home?color=blue`

Cu variabile de cale  
`http://localhost:8080/home/blue`

La implementarea pe server valoarea este extrasa din cale  
Devine mai complicat pentru mai mult de doua variabile

## Prin variabile de cale

Exemplu:

- `http://localhost:8080/home/blue`

```
@RequestMapping("/home/{color}")
public String home(
    @PathVariable String color,
    Model page) {
    page.addAttribute("username", "Katy");
    page.addAttribute("color", color);
    return "home.html";
}
```

Variabila de cale {color} reprezinta  
valoarea introdusa in cale

Adnotarea @PathVariable pentru  
parametrul metodei ia valoarea din cale

## Utilizarea metodelor HTTP GET si POST

### Metodele HTTP

- HTTP GET – cererea de la client pentru a obtine date de la server fara sa le modifice
- HTTP POST – cererea de la client care trimite noi date de adaugat la server
- HTTP PUT - cererea de la client care modifica o inregistrare de la server
- HTTP PATCH – cererea de la client care va modifica partial o inregistrare de la server
- HTTP DELETE – cererea de la client de a sterge anumite date pe server

Este incorect sa se utilizeze o metoda HTTP diferit fata de scopul pentru care a fost proiectata!

## Utilizarea metodelor HTTP GET si POST

### Exemplu aplicatie

- Aplicatia gestioneaza o lista de produse. Fiecare produs are nume si pret. Aplicatia afiseaza lista tuturor produselor si permite adaugarea unui nou produs la lista
- Cazuri de utilizare: arata toate produsele din lista (utilizare HTTP GET) si adaugarea unui produs la lista (utilizare HTTP POST)

## Utilizarea metodelor HTTP GET si POST

### Clasa **Produs**

- Se creeaza un package cu numele "**model**". In acest package se va crea clasa **Produs** cu attributele `nume` si `pret`

```
public class Produs {  
    private String nume;  
    private double pret;  
  
    // alte metode  
}
```

## Utilizarea metodelor HTTP GET si POST

### Clasa **ProdusService**

- Se creeaza un package cu numele "**service**". In acest package se va crea clasa **ProdusService** cu o lista – colectie de obiecte de tip **Produs** - si doua metode

```
@Service  
public class ProdusService {  
    private List<Produs> produse = new ArrayList<>();  
  
    public void addProdus(Produs p) {  
        produse.add(p);  
    }  
  
    public List<Produs> findAll() {  
        return produse;  
    }  
}
```

Discutia este simplificata pentru intelegerea metodelor HTTP



## Utilizarea metodelor HTTP GET si POST

### Clasa **ProdusController**

- Se creeaza un package cu numele "**controllers**". In acest package se va crea clasa **ProdusController** care va apela metodele implementate de clasa service

```
@Controller
public class ProdusController {
    private final ProdusService produsService;

    public ProdusController (ProdusService produsService) { ❶
        this. produsService= produsService;
    }
}
```

Injectie dependenta prin constructorul cu parametrii pentru a folosi obiectul service din contextul Spring

## Utilizarea metodelor HTTP GET si POST

### Clasa **ProdusController** (continuare)

- Afisarea intr-o pagina a listei de produse

```
@Controller
public class ProdusController {
    ...

    @RequestMapping("/produse") ❶
    public String viewProduse(Model model) { ❷
        var produse = produsService.findAll(); ❸
        model.addAttribute("produse", produse); ❹

        return "produse.html"; ❺
    }
}
```

- ❶ Se mapeaza actiunea controllerului pe calea /produse; @RequestMapping foloseste metoda HTTP GET
- ❷ Parametrul model trimite date catre view
- ❸ Se obtine lista de produse de la service
- ❹ Se trimite catre view lista de produse
- ❺ Se returneaza numele de view care va fi preluat de dispatcher

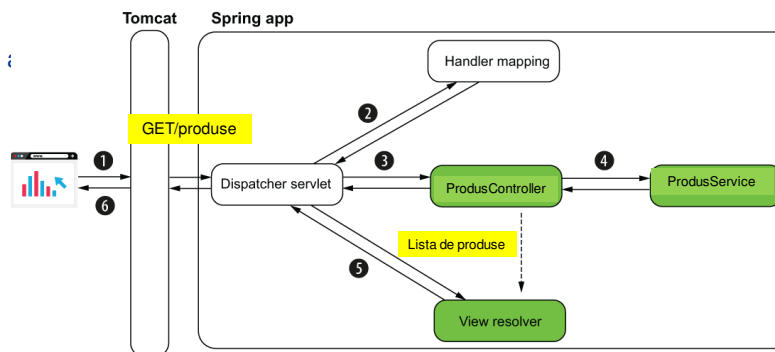
## Utilizarea metodelor HTTP GET si POST

Pagina HTML `produse.html` ia lista de produse de la controller si le afiseaza intr-un tabel HTML

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"> ❶
  <head>
    <meta charset="UTF-8">
    <title>Home Page</title>
  </head>
  <body>
    <h1>Produse</h1>
    <h2>Vizualizare produse</h2>
    <table>
      <tr>
        <th>NUME PRODUS</th> ❷
        <th>PRET PRODUS</th> ❷
      </tr>
      <tr th:each="p: ${produse}" > ❸
        <td th:text="${p.nume}"></td> ❹
        <td th:text="${p.pret}"></td> ❹
      </tr>
    </table>
  </body>
</html>
```

- ❶ prefixul "th" pentru utilizarea de capabilitati Thymeleaf
- ❷ defineste headerul static al tabelului
- ❸ th:each din Thymeleaf pentru a itera intr-o colectie si a afisa cate o linie de tabel pentru fiecare produs din lista
- ❹ afiseaza nume si pret pentru fiecare produs din lista

## Utilizarea metodei HTTP GET



- ❶ clientul trimite cererea HTTP GET prin calea /produse
- ❷ dispatcher servlet foloseste Handler mapping pentru a gasi la controller actiunea pentru calea /produse
- ❸ dispatcher servlet apeleaza actiunea controllerului
- ❹ controllerul cere lista de produse de la service pe care o trimite catre view
- ❺ view-ul este pus in raspunsul HTTP
- ❻ raspunsul HTTP este trimis clientului

## Utilizarea metodei HTTP POST

Clasa **ProdusController** (continuare)

- Adaugarea unui produs

```
@Controller
public class ProduseController {

    // ....

    @RequestMapping(path = "/produse",
                    method = RequestMethod.POST) 1
    public String addProdus(
        @RequestParam String nume, 2
        @RequestParam double pret, 2
        Model model
    ) {
        Produs p = new Produs(); 3
        p.setNume(nume); 3
        p.setPret(pret); 3
        produsService.addProdus(p); 3

        var produse = produsService.findAll(); 4
        model.addAttribute("produse", produse); 4

        return "produse.html"; 5
    }
}
```

1 Se mapeaza actiunea controllerului pe calea /produse; @RequestMapping foloseste metoda HTTP POST

2 Se folosesc parametrii din cale pentru atributele nume si pret ale unui produs

3 Se creeaza un obiect Produs cu atributele din cerere. Obiectul se adauga listei

4 Se obtine lista de produse care trimite catre view

5 Se returneaza numele de view care va fi preluat de dispatcher

37

## Utilizarea adnotari @GetMapping si @PostMapping

Clasa **ProdusController** (variantea cu aceste adnotari)

```
@Controller
public class ProduseController {

    // ....

    @GetMapping("/produse") 1
    public String viewProdus(Model model) {
        var produse = produsService.findAll();
        model.addAttribute("produse", produse);

        return "produse.html";
    }
}
```

1 @GetMapping mapeaza actiunea controllerului pe cererea HTTP GET cu calea /produse;

38

## Utilizarea adnotari @GetMapping si @PostMapping

Clasa **ProdusController** (varianta cu aceste adnotari)

- Adaugarea unui produs

```
@Controller
public class ProduseController {

    // ....

    @PostMapping("/produse")
    public String addProdus(
        @RequestParam String nume,
        @RequestParam double pret,
        Model model
    ) {
        Produs p = new Produs();
        p.setNume(nume);
        p.setPret(pret);
        produsService.addProdus(p);

        var produse = produsService.findAll();
        model.addAttribute("produse", produse);

        return "produse.html";
    }
}
```

2 @PostMapping mapeaza actiunea controllerului pe cererea HTTP POST cu calea /produse;

Aplicatii Web cu Suport Java, sem.I, 2024-2025

39

39

## Pagina HTML (modificari)

Pagina HTML produse.html permite utilizatorului sa apeleze actiunea controllerului pentru HTTP POST si adaugarea unui produs in lista

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8">
<title>Home Page</title>
</head>
<body>

<!-- .....-->

<h2>Adaugare produs</h2>
<form action="/produse" method="post">
    Name: <input
        type="text"
        name="nume"><br />
    Price: <input
        type="number"
        step="any"
        name="pret"><br />
    <button type="submit">Adauga produs</button>
</form>
</body>
</html>
```

1 Cand se submite, din HTML se face cererea POST pe calea /produse

2 Componenta input permite utilizatorului sa introduca numele produsului. Valoarea este trimisa prin parametru cererii cu cheia "nume"

3 Componenta input permite utilizatorului sa introduca pretul produsului. Valoarea este trimisa prin parametru cererii cu cheia "pret"

4 utilizatorul foloseste butonul submit pentru a submite formularul

Aplicatii Web cu Suport Java, sem.I, 2024-2025

40

40

## Utilizarea adnotari @GetMapping si @PostMapping

Clasa **ProdusController** (variantea cu aceste adnotari)

- Adaugarea unui produs (variantea)

```
@Controller
public class ProduseController {

    // ....

    @PostMapping("/produse")
    public String addProdus(
        Produs p,
        Model model
    ) {
        produsService.addProdus(p);

        var produse = produsService.findAll();
        model.addAttribute("produse", produse);

        return "produse.html";
    }
}
```

Numele parametrilor din cererea HTTP sunt aceiasi cu attributele clasei Produs. Spring va crea automat obiectul

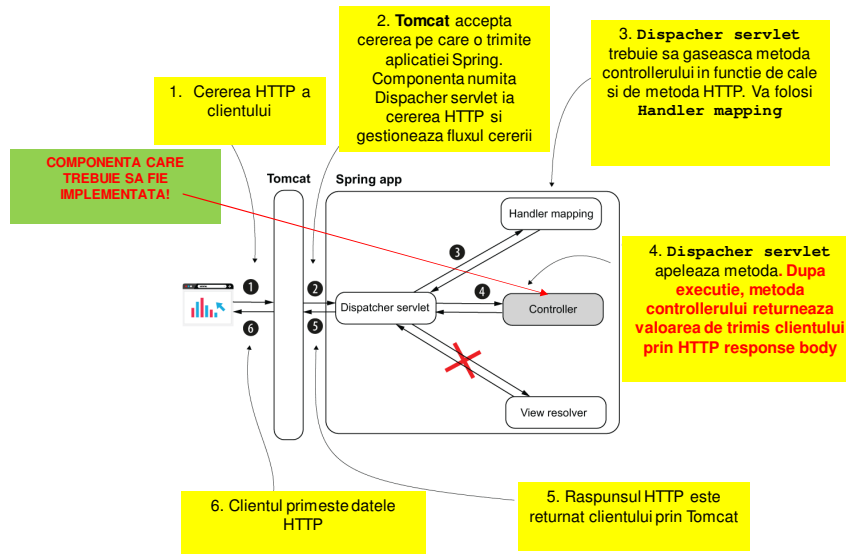
2 foloseste clasa Produs ca parametru al metodei controllerului. Clasa Produs trebuie sa aiba un constructor implicit;

## Implementarea serviciilor REST

REST = REpresentational State Transfer

- Servicii REST = modalitatea de a implementarea comunicarea intre doua aplicatii
- Aplicatia client poate apela solutia de backend prin endpoints REST

## Arhitectura Spring MVC la implementarea de endpoints REST



43

## Implementarea serviciilor REST

### Probleme de comunicare

- Daca actiunea controllerului dureaza mult, cererea HTTP la endpoint intra in timeout si se intrerupe comunicatia
- Trimiterea unui volum mare de date prin cererea HTTP poate intra in timeout si se intrerupe comunicatia.
- Prea multe apeluri concurente la un endpoint pun presiune pe componenta backend care poate pica
- Reteaua suporta toate apelurile HTTP, iar reseaua nu este 100% fiabila. Exista intotdeauna o sansa ca un apel la un endpoint sa pice din cauza acesteia.

44

## Implementarea serviciilor REST

### REST endpoint in clasa controller

```
@Controller ①
public class HelloController {

    @GetMapping("/hello") ②
    @ResponseBody ③
    public String hello() {
        return "Hello!";
    }
}
```

- ① **@Controller** marcheaza clasa ca fiind controller Spring
- ② **@GetMapping** asociaza metoda HTTP GET si o cale cu actiunea controllerului
- ③ **@ResponseBody** informeaza dispatcher servlet ca metoda nu returneaza un nume de view, ci direct un raspuns HTTP

## Implementarea serviciilor REST

Problema: **@ResponseBody** se repeta si devine cod duplicat

```
@Controller
public class HelloController {

    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello!";
    }

    @GetMapping("/salut")
    @ResponseBody
    public String salut() {
        return "Salut!";
    }
}
```

Solutie: adnotarea **@RestController** – este combinatia intre **@Controller** si **@ResponseBody**

## Implementarea serviciilor REST

@RestController Evita problema codului duplicat

```
@RestController ❶  
public class HelloController {  
  
    @GetMapping("/hello")  
    public String hello() {  
        return "Hello!";  
    }  
  
    @GetMapping("/salut")  
    public String salut() {  
        return "Salut!";  
    }  
}
```

❶ @RestController – este înlocuitor pentru @Controller

## Finalizare

**Lucrare de verificare finala** – 20% din nota finala; prezenta este obligatorie pentru a promova disciplina;

Planificare pe 14 ianuarie 2025; ora 14:00;

Organizarea in serii va fi anuntata pe moodle !!!



## Continuare utilizare de cunostinte de Java

Cursul "Programare in timp real" anul IV, Pachetul 2A4 include java multithreading, java IO, java networking, etc